REFINING THE DOMAIN MODEL pdf

1: Creating a New Model

Review Questions 18 -- Refining the Domain Model; Review UML Notation. Draw diagrams showing each of these facts using the UML A is associated with B.

It not only contains a set of technical ideas, but it also consists of techniques to structure the creativity in the development process. The key of Domain-Driven Design is understanding the customers needs, and also the environment in which the customer works. The problem which the to-be-written program should solve is called the problem domain, and in Domain-Driven Design, development is guided by the exploration of the problem domain. While talking to the customer to understand his needs and wishes, the developer creates a model which reflects the current understanding of the problem. This model is called Domain Model because it should accurately reflect the problem domain of the customer. Then, the model is refined again â€" and the whole process of discussion with the customer starts again. Thus, Domain-Driven Design is an iterative approach to software development. Still, Domain-Driven Design is very pragmatic, as code is created very early on instead of extensive requirements specifications ; and real-world problems thus occur very early in the development process, where they can be easily corrected. Normally, it takes some iterations of model refinement until a domain model adequately reflects the problem domain, focusing on the important properties, and leaving out unimportant ones. In the following sections, some core components of Domain-Driven Design are explained. It starts with an approach to create a ubiquitous language, and then focuses on the technical realization of the domain model. After that, it is quickly explained how Flow enables Domain-Driven Design, such that the reader gets a more practical understanding of it. Note We do not explain all details of Domain-Driven Design in this work, as only parts of it are important for the general understanding needed for this work. More information can be found at [Evans]. For instance, the customer is an expert in his business, and he wants to use software to solve a certain problem for him. Thus, he has a very clear idea on the interactions of the to-be-created software with the environment, and he is one of the people who need to use the software on a daily basis later on. Because he has much knowledge about how the software is used, we call him the Domain Expert. On the other hand, there are the developers who actually need to implement the software. While they are very skilled in applying certain technologies, they often are no experts in the problem domain. Now, developers and domain experts speak a very different language, and misconceptions happen very often. To reduce miscommunication, a ubiquitous language should be formed, in which key terms of the problem domain are described in a language understandable to both the domain expert and the developer. Thus, the developers learn to use the correct language of the problem domain right from the beginning, and can express themselves in a better way when discussing with the domain expert. Furthermore, they should also use the ubiquitous language throughout all parts of the project: Not only in communication, design documents and documentation, but the key terms should also appear in the domain model. Names of classes, methods and properties are also part of the ubiquitous language. By using the language of the domain expert also in the code, it is possible to discuss about difficult-to-specify functionality by looking at the code together with the domain expert. This is especially helpful for complex calculations or difficult-to-specify condition rules. Thus, the domain expert can decide whether the business logic was correctly implemented. Creating a ubiquitous language involves creating a glossary, in which the key terms are explained in a way both understandable to the domain expert and the developer. This glossary is also updated throughout the project, to reflect new insights gained in the development process. Usually, UML is employed for that, which just contains the relevant information of the problem domain. The domain model consists of objects as DDD is a technique for object-oriented languages , the so-called Domain Objects. There are two types of domain objects, called Entities and Value Objects. If a domain object has a certain identity which stays the same as the objects changes its state, the object is an entity. Otherwise, if the identity of an object is only defined from all properties, it is a value object. We will now explain these two types of objects in detail, including practical use-cases. Furthermore, association mapping is explained, and aggregates are introduced as a way to further structure the code. For example, a user can have a user name as identity, a student a matriculation ID.

Although properties of the objects can change over time for example the student changes his courses , it is still the same object. Thus, the above examples are entities. The identity of an object is given by an immutable property or a combination of them. In some use-cases it can make a lot of sense to define identity properties in a way which is meaningful in the domain context: If building an application which interfaces with a package tracking system, the tracking ID of a package should be used as identity inside the system. Doing so will reduce the risk of inconsistent data, and can also speed up access. For some domain objects like a Person, it is highly dependent on the problem domain what should be used as identity property. In an internet forum, the e-mail address is often used as identity property for people, while when implementing an e-government application, one might use the passport ID to uniquely identify citizens which nobody would use in the web forum because its data is too sensible. In case the developer does not specify an identity property, the framework assigns a universally unique identifier UUID to the object at creation time. It is important to stress that identity properties need to be set at object creation time, i. As we will see later, the object will be referenced using its identity properties, and a change of an identity property would effectively wipe one object and create a new one without updating dependent objects, leaving the system in an inconsistent state. In a typical system, many domain objects will be entities. However, for some use-cases, another type is a lot better suited: Value objects, which are explained in the next section. Integer, float, string, float and array. However, it is often the case that you need more complex types of values inside your domain. These are being represented using value objects. The identity of a value object is defined by all its properties. Thus, two objects are equal if all properties are equal. For instance, in a painting program, the concept of color needs to be somewhere implemented. A color is only represented through its value, for instance using RGB notation. If two colors have the same RGB values, they are effectively similar and do not need to be distinguished further. Value objects do not only contain data, they can potentially contain very much logic, for example for converting the color value to another color space like HSV or CMYK, even taking color profiles into account. Thus, value objects are immutable. For example, there might be a method mix on the Color object, which takes another Color object and mixes both colors. Still, as the internal state is not allowed to change, the mix method will effectively return a new Color object containing the mixed color values. As value objects have a very straightforward semantic definition similar to the simple data types in many programming languages , they can easily be created, cloned or transferred to other subsystems or other computers. Furthermore, it is clearly communicated that such objects are simple values. Internally, frameworks can optimize the use of value objects by re-using them whenever possible, which can greatly reduce the amount of memory needed for applications. Entity or Value Object? An example illustrates this: For many applications which need to store an address, this address is clearly a value object - all properties like street, number, or city contribute to the identity of the object, and the address is only used as container for these properties. However, if implementing an application for a postal service which should optimize letter delivery, not only the address, but also the person delivering to this location should be stored. This name of the postman does not belong to the identity of the object, and can change over time â€” a clear sign of Address being an entity in this case. So, generally it often depends on the use-case whether an object is an entity or value object. People new to Domain-Driven Design often tend to overuse entities, as this is what people coming from a relational database background are used to. So why not just use entities all the time? The technical answer is: Domain objects have relationships between them. In the domain language, these relations are expressed often as follows: These relations are called associations in the domain model. In the real world, relationships are often inherently bidirectional, are only active for a certain time span, and can contain further information. However, when modelling these relationships as associations, it is important to simplify them as much as possible, encoding only the relevant information into the domain model. Especially complex to implement are bidirectional many-to-many relations, as they can be traversed in both directions, and consist of two lists of objects which have to be kept in sync manually in most programming languages such as Java or PHP. Still, especially in the first iterations of refining the domain model, many-to-many relations are very common. The following questions can help to simplify them: Is the association relevant for the core functionality of the application? If it is only used in rare use cases and there is another way to receive the needed information, it is often better to drop the association

altogether. For bidirectional associations, can they be converted to unidirectional associations, because there is a main traversal direction? Traversing the other direction is still possible by querying the underlying persistence system. Can the association be qualified more restrictively, for example by adding multiplicities on each side? The more simple the association is, the more directly it can be mapped to code, and the more clear the intent is. However, often it is the case that certain objects are parts of a bigger object. For example, when modeling a Car domain object for a car repair shop, it might make sense to also model the wheels and the engine. As they are a part of the car, this understanding should be also reflected in our model. Such a part-whole relationship of closely related objects is called Aggregate. An aggregate contains a root, the so-called Aggregate Root, which is responsible for the integrity of the child-objects. Furthermore, the whole aggregate has only one identity visible to the outside: The identity of the aggregate root object. Thus, objects outside of the aggregate are only allowed to persistently reference the aggregate root, and not one of the inner objects. For the Car example this means that a ServiceStation object should not reference the engine directly, but instead reference the Car through its external identity.

REFINING THE DOMAIN MODEL pdf

2: Manning | Functional and Reactive Domain Modeling

When we get back to refining the Domain Model in later chapters, I won't use such a TDD-ish way of describing all the tiny design decisions as we go. It will be a more condensed description. But let's end the chapter with a discussion about another style of API.

Enabler features may also result in a change of responsibilities or may introduce new value objects. Relationships drive both effective requirements definition and design decisions see the example below in figure 5 and the associated description. Relationships in a domain model can be pretty standard e. When defining the relationships it is much more important to adequately capture real connections between the entities that convey the meaning of their role rather than to follow format agreements indiscriminately. The common language resulting from domain modeling is used at all levels of the Agile organization to foster unambiguous shared understanding of the problem domain, requirements, and architecture â€" see Figure 3. Common Language Used Throughout Agile Organization Common language â€" even though is crucial to the product development â€" has natural limitations that every organization should be aware of. For example, the language of marketing materials may sometimes use terms that diverge from the common language, in order to emphasize certain temporal or subjective aspects associated with current market trends or challenges. Teams that use ATDD inevitably use common language in their specification workshops when defining human readable tests. Alternative Approaches Domain modeling is not only useful for analysis but is often a good conceptual model for the system design. See [2] for a systematic and detailed outline of such best practices, known by the term of Domain-Driven Design. This approach provides a natural and very effective way of managing the inherent complexity of software development that is vital at large scale. Figure 4, adapted from [3], chapter 2, shows a comparison of effort spent on enhancing software functionality versus complexity by different approaches: When design is based on the domain When design is based on data structure or transaction scripts. A sense of the relationship between domain-oriented and data- or transaction-centric approaches. Large scale software solutions almost inevitably have complex domain logic. Thus data- or transaction-centric design approaches imply very high cost of maintenance. Nevertheless, too many organizations end up with highly complex system designs that imply a lot of effort to enhance the system. While in some cases such approaches may make sense â€" and we will discuss those below â€" most often such a design in reality is based on personal preferences of the system architects and teams rather than on business drivers. One of the many reasons to base system design on the domain structure is to foster reasonable usage of patterns that support maintainability and enable highly incremental, concurrent development. So, in our example of the subscription management system, domain modeling and requirements may logically suggest that subscription methods will represent the primary source of change. Thus, given the different scenarios for opt-in and opt-out functionality, it seems quite logical to use a Bridge Pattern, shown in figure 5 to isolate the area of frequent change and reduce the number of entities in the system â€" see [4], Appendix B. A bridge pattern derived from analysis of the domain model This is just one example of how domain modeling can be effectively used for Commonality-Variability Analysis CVA to foster effective system object models. See [5], chapter 8 for more detail on the CVA method. Domain Modeling, System Design, and Nonfunctional Requirements Nonfunctional Requirements , on the other hand, represent the primary reason to build system design around data structure or transaction scripts rather than the domain model. Typically NFRs like performance or scalability may result in cases where domain logic is spread across a bunch of large SQL-scripts, or where too much logic is in the client-side validation scripts, and so on. Even though the use of such a Transaction Script approach see [3], chapter 9 can be legitimate in certain cases, it should be used as an exception rather than the rule. A very few properly implemented exceptions will still allow the Agile enterprise to benefit from a domain-oriented approach. Refactoring the Model Developing shared understanding with the help of domain modeling is an incremental process just like developing code that implements the underlying domain logic. This means that just like the code, the domain model is also subject to refactoring as our knowledge about the system improves, and as new domain entities and their

relations actualize, as table 1 suggests. In the Domain-Driven Design approach, keeping the system design and the current understanding of the problem domain up to date is relatively easy, and refactoring of both typically happens synchronously or nearly so see [2], part III. Designing the effective domain model is both an art and a science. Not uncommonly great insights about the structure and associations in the domain model emerge eventually. However, it is never late to start building the right understanding and to start gradually improving the code towards it â€" as new functionality allows â€" to be able to control the complexity. Summary Domain modeling is a great tool for agile enterprise to carry out a common language and a fundamental structure important for the analysis of features and epics. Domain model is defined and continuously refactored as enterprise knowledge about the domain improves and the system functionality evolves. Domain model serves a vital link between the real world where the problem domain resides and the code â€" domain-oriented design approaches allow to control rapidly growing complexity and cost of maintenance and enhancement effort. Domain modeling is highly collaborative and visual effort that involves system architects, product management, stakeholders and teams all working towards better shared understanding of the priorities and better ways to implement them. Thus, if you only model one thing, model the domain. Learn More [1] Ambler, Scott. Cambridge University Press, Tackling Complexity in the Heart of Software. Patterns of Enterprise Application Architecture.

REFINING THE DOMAIN MODEL pdf

3: Agile Software Development with ICONIX Process

The regional domain model consists of a forest root domain and one or more regional domains. Creating a regional domain model design involves identifying what domain is the forest root domain and determining the number of additional domains that are required to meet your replication requirements.

Nonblocking API design with futures 6. Asynchrony as a stackable effect 6. Monad transformer-based implementation 6. Task as the reactive construct 6. Explicit asynchronous messaging 6. A sample use case 6. A graph as a domain pipeline 6. Domain models and actors 6. Modeling with reactive streams 7. The Reactive Stream Model 7. When to use the Stream Model 7. The Domain Use Case 7. Stream based Domain Interaction 7. The Back Office 7. Major Takeaways from the Stream Model 7. Making Models Resilient 7. Supervision with Akka Streams 7. Clustering for Redundancy 7. Reactive persistence and event sourcing 8. Persistence of Domain Models 8. Separation of Concerns 8. Event Sourcing events as the ground truth 8. Implementing an Event Sourced Domain Model functionally 8. Events as First Class Entities 8. Commands as Free Monads over Events 8. The Event Store 8. Summary of the Implementation 8. Other Models of Persistence 8. Manipulating Data Functionally 8. Reactive Fetch that pipelines to Akka Streams 8. Testing your domain model 9. Designing testable domain models 9. Decoupling side effects 9. Custom interpreters for domain algebra 9. Parametricity and testing 9. Revisiting the algebra of your model 9. Property based testing 9. Verifying properties from our domain model 9. Better than xUnit based testing? Rehashing the core principles for functional domain modeling Think in expressions Abstract early, evaluate late Use the least powerful abstraction that fits Publish what to do, hide how to do within combinators Decouple algebra from the implementation Isolate bounded contexts Prefer futures to actors To capture their dynamic relationships and dependencies, these systems require a different approach to domain modeling. A domain model composed of pure functions is a more natural way of representing a process in a reactive system, and it maps directly onto technologies and patterns like Akka, CQRS, and event sourcing. About the book Functional and Reactive Domain Modeling teaches you consistent, repeatable techniques for building domain models in reactive systems. This book reviews the relevant concepts of FP and reactive architectures and then methodically introduces this new approach to domain modeling.

REFINING THE DOMAIN MODEL pdf

4: Oracle - Object-Oriented Analysis and Design Using UML

Domain modeling is one of the key models used in software engineering: if you only model one thing in Agile, model the domain. A relatively small domain-modeling effort is a great tool for controlling complexity of the system under development.

As an iterative development framework, the Rational Unified Process,or RUP, is flexible enough to suit a variety of project management styles. Building solutions using model-driven architecture MDA methods requires changes to the development process. While our experience has been that many of the current best practices for enterprise software development are still applicable, a model-driven approach to the development process requires some important changes to those practices. It provides a disciplined approach to assigning tasks and responsibilities within a development organization and has been applied to projects of varying size and complexity, with small teams and large, and with durations lasting weeks to years. Figure 1 illustrates the overall architecture of the RUP in two dimensions. The horizontal axis represents time and shows the lifecycle aspects of the process. The management perspective of lifecycle phases is shown across the top, and the software engineering and project management perspective of iterations is displayed along the bottom. The vertical axis represents disciplines grouped logically, showing the static aspect of the process -- how RUP is described in terms of process components, disciplines, activities, workflows, artifacts, and roles. The RUP concepts of disciplines, phases, and iterations At any point in time during a RUP-based project, there is activity taking place in a variety of disciplines. What distinguishes one lifecycle phase from another is not the total absence of a discipline, but the relative amount of contribution that discipline is making in the overall work streams. The mix of activities varies over time as the emphasis and priorities of the project change. For example, in early iterations you spend more time on requirements; in later iterations you spend more time on implementation. RUP phases and iterations From a management perspective, the RUP software lifecycle consists of four sequential phases, each concluded by a major milestone. An assessment is performed to determine whether the objectives of the phase have been met. A satisfactory assessment allows the project to move to the next phase. Briefly, the phases of a RUP lifecycle are: The goal of the Inception phase is to reach agreement among all stakeholders on the lifecycle objectives for the project. Typically, there are significant business and requirements risks that must be addressed before the project can proceed. For projects focused on small enhancements to an existing system, the Inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do. The Lifecycle Objectives Milestone is the primary exit criteria of Inception. It evaluates the basic viability of the project. The goal of the Elaboration phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the Construction phase. Elaboration produces an executable system that marries the essential business requirements with the technical architecture and demonstrates the viability of the technical approach chosen. The architecture evolves out of a consideration of the most significant requirements those that have a great impact on the architecture of the system and an assessment of risk. The Lifecycle Architecture milestone establishes a managed baseline for the architecture of the system and enables the project team to scale during the Construction phase. The goal of the Construction phase is clarifying the remaining requirements and completing the development of the system based upon the baselined architecture. The Construction phase is in some sense a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mindset undergoes a transition from the development of intellectual property during Inception and Elaboration to the development of deployable products during Construction and Transition. The Initial Operational Capability milestone determines whether the product is ready to be deployed into a user acceptance environment. The focus of the Transition phase is to ensure that software is available for its end users. The Transition phase can span several iterations; it includes testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback should focus mainly on fine-tuning the product, as well as configuring, installing, and addessing usability issues; all the major structural issues should have been

worked out much earlier in the project lifecycle. The Product Release Milestone is the point at which you decide if the objectives of the project were met, and if you should start another development cycle. From a software engineering and project management perspective, things happen day-to-day based on iterations. Phases consist of multiple iterations -- distinct sequences of activities with a base-lined plan and valuation criteria resulting in a release of artifacts internal or external. Every phase of the lifecycle consists of a series of iterations and the character of the iterations differs depending on where you are in the lifecycle. This is not surprising; RUP reflects current industry best practice and typically does not codify approaches until they are well established within the field. Our experiences, however, indicate a number of important ways in which we can enhance RUP with best practices for MDA projects. In particular, the backbone of RUP, an architecture-centric and iterative development process, is highly consistent with MDA concepts and offers an excellent basis for succeeding with MDA. However, there are some areas where additional guidance on MDA is appropriate. Elaboration is the main phase impacted by an MDA project. It is important to look at Elaboration activities and briefly describe MDA modifications. The Architect role is the main role in Elaboration that requires additional consideration. Essentially, this role defines specific MDA activities and artifacts, creates the transformations, and so on. MDA is as much about model-driven automation as it is about model-driven architecture. Rather, the standard activities in each role will require details to be added regarding how those activities are automated using the specific tools and MDA techniques chosen. For example, a team working on object-relational data mapping may be using a modeling tool with a set of MDA transformations. But at a high level their workflow related to "define design classes" will largely remain the same. More pervasively, the primary changes to RUP consist of a more subtle change of perspective on the development process. MDA encourages architects and developers to work at higher levels of abstraction than typically expected in non-MDA projects. This is most apparent during construction where the code automation aspects of MDA significantly change the emphasis of the implementation tasks. Developers are able to continue to work with more abstract analysis and design models as inputs to MDA transformations. They find that they work less with actual implementation models and source code, and more with designs for the appropriate business-focused workflow of the solution. A smaller subset of developers will be implementing the model-to-code transformations themselves. MDA transformations are frequently constructed around pre-built solution frameworks also called "Reference Architectures" or "Application Frameworks". The MDA transformations augment these frameworks with domain-specific business logic. This approach of coupling MDA and solution frameworks makes a great deal of sense since the entire MDA approach is centered on automating a set of repeatable techniques and assets. However, it further changes the development process toward greater emphasis on reuse, management assets, and incremental delivery of solutions. Rather, they are created by extending an existing solution framework with domain-specific business logic, by connecting to and manipulating information from different sources, and by designing context-rich user display and interaction services. Hence, the development approach that is followed is not the classic "waterfall" scenario, where requirements gathering is followed by analysis and design, leading to the implementation of the system. Rather, it is one of continual extension and refinement of an existing partial solution toward a desired goal through a set of iterations that add value to the solution. These partial solutions that form the heart of a new system may come from one of several sources: An existing set of applications. The primary approach may be to take an existing solution and extend it in useful ways, as dictated by the business need. Hence, much of the design work involves understanding how those existing applications are architected, and where meaningful extensions can be added without compromising the qualities of the existing applications. In most cases, the process is complicated by the fact that the original applications were not designed with the goal of reuse in mind. A proprietary application framework used by the organization. Having built many kinds of similar solutions in a particular domain, some organizations have extracted core application capabilities as reusable proprietary services to be employed in future solutions. These services help to improve productivity of a family of systems that share common characteristics, and increase the predictability of future system development. An acquired application framework, whether open source or commercial in origin. Recognizing the consistent architectural patterns that are used in designing certain kinds of applications has led to a number

of technologies to help organizations create solutions conforming to those patterns. The resulting application frameworks are available both commercially and in the open source community, and are delivered as standalone frameworks or bundled with tools that help create, manage, and extend those frameworks. A set of extensions and customizations to packaged applications. Many organizations acquire comprehensive solutions for key business processes from packaged application vendors. However, organizations typically need to customize those solutions to meet their needs. As a result, the packaged application vendors 1 have structured their solutions to support different kinds of extension and customization, 2 offer well-defined APIs to access internals of the packaged application, or 3 augment the packaged applications with detailed design documents, extension examples, and package-specific tools. Given these possibilities, the primary task faced by many IT project managers is to create a clear understanding of their domain, to express that understanding in a platform-independent domain model supporting various kinds of analysis to ensure its correctness and consistency, and to map that domain model to a platform-specific implementation realized by extending the solution framework. Model-to-model transformations help in refining the domain, while model-to-code transformations map the domain model to the specific solution framework. In model-to-code transformations, the solution framework plays a key role, because it constrains and guides the kinds of transformations that are meaningful. You can create a set of transformations based on that knowledge. Indeed, you can created wizard-driven tooling to automate creation of those transformations for domain models containing appropriate kinds of information. More generally, by using a solution framework as the basis for a system, the task of writing model-to-code transformations is significantly eased, and we gain greater efficiency, predictability, repeatability, and manageability of the resulting solutions. In summary, we reiterate that software is rarely created from scratch, and that model transformations to another model or to code help leverage existing solution frameworks. As this approach of building on existing software increases, the role of MDA and its value increases by helping to automate how we extend and customize those frameworks. Some view this "custom automation" as the only viable option for creating systems of increasing complexity, and in response to the constraints placed on us by the components and technologies we deploy. Summary and future directions Model-driven approaches to software and system development have been in use for some time. MDA techniques enable organizations to construct custom automations for model-to-model and model-to-code transformations. Using these transformations, technical experts can share their expertise across a large development team. In particular, MDA offers a number of advantages over other approaches: Productivity of development is increased by helping to insulate a majority of developers from technical details that they do not need to consider to carry out their task of designing enterprise solutions to meet a business need. More time is spent focusing on their task at hand: MDA improves quality by encouraging reuse of known patterns of behavior, building on existing architectural designs, and leveraging expertise more effectively. The use of automation also promotes consistency and improves the quality of the system design and implementation, particularly as the solution evolves through maintenance and upgrade. The use of automation speeds up development, particularly when many of the tasks of the developer are repetitive and cumbersome. These experiences reveal that, while traditional design and implementation practices are relevant to MDA projects, there are a number of additional requirements that must be addressed to ensure that the approach is optimally applied. We have described many of these requirements and illustrated them with practical examples. In Part 2, we distilled our key findings into 12 lessons for practical application of MDA. These lessons, however, are not specific to a single set of technologies. They have also been applied within other IBM Rational tools.

REFINING THE DOMAIN MODEL pdf

5: An introduction to model-driven architecture

Larman - Part5 - Elaboration: Iteration 3 Chapter 27 - Refining the Domain Model Objectives. Add association classes to Domain Model; Add aggregation relationships.

Here is the relevant content from the email: Entity is a business concept that exposes behavior. A collection of entities may have different behavior varied upon the type of aggregate that encapsulates it? Or is this over-thinking it? How you might model this as entities and value objects? And is there an aggregate concept lurking in there? The line items would likely be value objects, since its their properties that probably matter more than trying to preserve identity over time for them. Unlike the agile classes I teach, which are one inclusive price, other companies offering training and coaching might want to break out the costs to rent a venue, pay for catering, cover instructor travel expenses, coaching days etc. So purchase orders would need to handle multiple line items in many cases. This is fairly standard for POs anyway. Aggregates Support Higher Level Concepts The business could work with LineItems individually, but in practice never would, since they only really make sense in light of their PurchaseOrder. And you would mostly likely want to work with the higher-level concept of Purchase Orders rather than always having to deal at the granularity of the Line Items. I could imagine business rules for certain types of Purchase Orders that the sum of the values of the individual Line Items could not exceed a certain amount for the Purchase Order to be approved. A Purchase Order would probably need to have at least one Line Item to be valid. Here is a case of two or more objects that seem to belong together most of the time in terms of how you need to work with them. And make the PO entity the root of the aggregate. So what we have in this example is an aggregate consisting of a single entity, the Purchase Order functioning as the root of the aggregate , and a set of one or more associated Line Item value objects. Aggregates, Invariants and Consistency This aggregate boundary is part of the model, and allows us to enforce invariants e. Between POs we can have eventual consistency, since we are comfortable with not trying to keep all our aggregates in sync with each other. In terms of how this plays out, you would typically have a repository for persisting and retrieving the PO aggregates. When you get a PO from its repository, it would be created whole, with all its Line Items as well. As an aside, this is what makes document stores a nice fit for aggregates, since aggregate and document boundaries often tend to align in terms of how a model is used. For background reading, see the DDD Reference book, especially pp. Aggregate Boundaries and Behavior I love concrete examples. This actually demonstrates my confusion between Aggregates and Root Entities. Namely, that aggregates represent a collection of behaviors that are transactionally bound and express the domain model. So as more behavior needs to be added to this PO, I would try to model that as behaviors on new or existing value objects where possible. Additionally, I feel that an Aggregate Root concept is a vestigial idea posited by Eric to appease some OO or implementation concern. I find the aggregate root concept helpful though, since a single entity typically takes that responsibility. Yes, if I understand you correctly. As a concrete example, a PO has several behaviors that may not change as a training organization moves from inferiority to not-as-optimal. The design and implementation may currently consider entities like "Location Rental" or "Travel Expenses" to express the PO model; but after maturity of the model and company they choose to represent their line items more succinctly or with less inferior concepts. Aggregate boundaries may, and likely will, change over time as the model matures. Maybe the team realizes that Location Rental needs to be its own aggregate, for example. And if a PO is canceled then the Location Rental needs to be canceled too. To your point, though, the entities, value objects and domain events inside the aggregate could potentially change without affecting the aggregate boundary. Cargo is the aggregate root, with several value objects handling the business rules. Delivery essentially functions as a read projection of the Handling Event history. Cargo is focused on identity and not much else. All the interesting business logic is in the value objects.

REFINING THE DOMAIN MODEL pdf

6: CiteSeerX â€” Refining Incomplete Planning Domain Models Through Plan Traces

In addition, the process of documenting, reviewing and refining this text would cost a great deal more than asking our business customers to sign the whiteboard with the confidence that the domain model / use cases (i.e. requirements) covered on it have been captured correctly.

Each of the unit categories provides a selection of both Imperial units and SI units. If the Length unit is meters, the Mass unit must be kilograms. Likewise, if the Length unit is feet, the Mass unit must be pounds. Visual MODFLOW will internally convert your units for run purposes if they do not match the requirements, and the output will be converted back to your specified units. Flow Options The following Flow Option window allows you to configure all options related to the Flow model. The Flow Option window is divided into several sections: The Project Info frame is a Read-Only frame containing information about your project which you defined previously The Time Option frame allows you to set Date, Time, and Run information The Default Parameters frame allows you to set the default flow parameters for your model Time Option The Start Date of the model is the date corresponding to the beginning of the simulation. Currently, this date is relevant only for transient flow simulations where recorded field data may be imported for defining time schedules for selected boundary conditions Constant Head, River, General Head and Drain. The Start Time of the model is the time corresponding to the beginning of the simulation. Currently, this time is relevant only for transient flow simulations where recorded field data may be imported for defining time schedules for selected boundary conditions Constant Head, River, General Head and Drain. If the Steady State Flow option is selected, Visual MODFLOW will prepare the data set for a steady-state flow simulation, and will automatically use the data from the first stress period of each boundary condition and pumping well defined in your project. You must enter a valid number 1 is acceptable if you do not have a preference before proceeding. This parameter is not used if you have selected Transient Flow, however a number must still be entered. Although the simulation will always be run to the same equilibrium solution in Steady State, the total amount of water passing through boundary conditions i. For example, Zone Budget analysis of a Steady State solution would be affected by the simulation time, whereas regional head values would not. Additionally, this number would be important if you were performing a Well Optimization on your project. Default Parameters The default parameter values are considered reasonable values to be assigned as initial estimates of the material properties in the model. Please NOTE that once you have created the model, you will be able to enter new parameter values, and create varying parameter zones. Once you have edited the variables in the Flow Option window, click [Next] to proceed. Transport Options If you chose in Step 1 to run a Transport simulation, or if the Flow Engine you selected requires a Transport simulation, the Transport Option window will appear similar to the following screen capture. If you decide at a later date to add a Transport simulation to your project, you may do so by clicking on Setup-Edit Engines from the Main Menu. The Transport Option window is divided into several sections: The Project Info frame is a Read-Only frame containing information about your project which you defined previously The Default Dispersion Parameters frame allows you to enter Dispersivity and Diffusion values The Variant Parameters frame allows you to enter information for your first Transport Variant, including Sorption and Reaction options, and Species information. Default Dispersion Parameters The default parameter values are considered reasonable values to be assigned as initial estimates of the material properties in the model. Variant Parameters The Variant Title and Description are optional user-defined references for each transport variant. To the right of the Variant Title field is the internal Variant Number e. Depending upon the selected Transport Engine, you will also have the option of choosing from a number of available Sorption methods and Reactions. The Are the reaction parameters constant or spatially variable? If you are using an RT3D transport engine, you will have the option to select between Constant, and Spatially Variable, reaction parameters. The Species, Model Params, and Species Params tabs allow you to modify specific settings for each species, and for your Transport simulation. The New Species and Delete Species buttons allow you to add and remove species to your Transport simulation. This option is available only for certain transport engines. The definition of such user-defined reaction networks requires some basic understanding of

geochemical processes and concepts as well as some knowledge of the geochemical code PHREEQC. The Temperature button allows you to add a temperature species to your transport simulation. Once you have edited the variables in the Transport Option window, click [Next] to proceed. Model Domain The following Model Domain window allows you to configure all options related to the default model grid. The Model Domain window is divided into several sections: The Background Map frame allows you to select a site map The Grid frame allows you to specify the dimensions of the Model Domain and the Finite Difference Grid Background Map The Import a site map option is used to select a site map to overlay on the model domain. SHP files, and Graphics Image formats such as. Use the [Browse] button to locate and select the site map to include in the model. When creating a new model, only one sitemap may be used to setup the model grid. If a sitemap is imported for the model you will be able to align the model in the desired orientation on the map. For a detailed description on Importing Site maps and georeferencing images, see "Importing Site Maps" on page Otherwise the Xmin, Xmax, Ymin and Ymax co-ordinates must be entered. Instructions on refining the finite difference grid and assigning the model properties and boundary conditions are provided in Chapter 4. Grid The Grid frame contains options for the model dimensions and initial grid discretization. The initial grid will have uniform grid cell sizes throughout the entire model, but all grid spacing and elevations are customizable through the Visual MODFLOW interface. If your project requires a grid larger than the default limits, you may contact the Technical Support department at sws-support slb. Selecting the Model Region Selecting a model region involves the following tasks: In the Select Model Region window, the model domain is represented by an outline of a box with circular nodes at each corner and with arrows pointing along the X and Y axes see following figure. The model domain box can be shifted, expanded, and rotated to any alignment on the site map using the toolbar options described below. These options are also available under the View menu item. Typically, the model domain is aligned along the principal direction of groundwater flow and the domain is large enough to accommodate reasonable boundary conditions for the model. In situations where the model domain is rotated, the model will use a dual co-ordinate system whereby locations in the model may be expressed in either world co-ordinates or model co-ordinates see "Dual Co-ordinate Systems" on page Click-and-drag the mouse to select the zoom area. View the entire map display area. Click-and-drag a corner of the model domain box to stretch or shrink the size of the model domain. Double clicking on the model region will switch the mode between Resize Region and Rotate Region. Click-and-drag a corner of the model domain box to rotate and align the model domain. Double clicking on the model region will switch the mode between Rotate Region and Resize Region. Align the model domain with the x-axis. Enlarge the model domain to the full extents of the base map. Load a new file for the site map. SHP file is used for the site map, the top right-hand section will contain the Display Area co-ordinates where X1, Y1 is the map co-ordinate in the lower left-hand corner and X2, Y2 is the map co-ordinate in the top right-hand corner. If a graphics file is used for the site map, the top right-hand section will contain the co-ordinates of the two Georeference Points. These values cannot be modified unless one of the georeference points is deleted and a different georeference point is assigned. The Angle is the degree angle of rotation of the model region from the world co-ordinate axes. The Model Origin is the X and Y world co-ordinate location of the bottom left-hand corner of the model domain. The Model Corners are the model co-ordinates of the bottom left-hand corner X1, Y1 and the top right-hand corner X2, Y2 of the model domain. If the Angle is 0. However, if the model domain is rotated from the world co-ordinate axes, then the model will use a dual co-ordinate systems. The NRows and NColumns fields contain the number of finite difference rows and columns, respectively, for the model grid. In some situations e. In these cases, the attributes of the graphics image will not be aligned with similar attributes from a. DXF map or a. File Attributes Once the model domain location, orientation, and size have been selected, the File attributes window will appear as shown in the following figure.

REFINING THE DOMAIN MODEL pdf

7: CSE Review Questions 18 -- Refining the Domain Model

Explore techniques for refining the conceptual model by between the technical and domain experts. Learn from practical examples implemented in C# www.enganchecubano.com Understand the importance of focusing on the core domain and domain logic of your business.

First part to implement, parts of Order and OrderFactory Note The first few steps are necessary to properly create the domain objects. We must complete these steps before we can get to the interesting and crucial part, namely testing and implementing the logic that our model facilitates. CreateOrder new Customer ; Assert. This means we are free to name the tests the way we want. This also means we get the tech-talk out of the way and can focus the discussion on domain concepts and design. I like naming the tests as statements of what the tests intend to verify. That way, the collection of test names becomes a list of what should be possible and what should not. At first, the test looks extremely simple. To make the test compile, I need to create an empty Order class. Next, I need to create an OrderFactory class with a single method, but our problems start cropping up already when writing that method. One of the rules defined in Chapter 4 was that an Order always has to have a certain Customer, and this is solved by giving that responsibility to the OrderFactory, giving the Order a Customer at creation time. Therefore, the CreateOrder method of the OrderFactory needs to receive a Customer instance as a parameter which you can see in the test code. No rocket science there, but we have just started working on the Order Aggregate and we would like to forget about everything else right now. The first solution that springs to mind which is also visible in the test I wrote previously would be to also just create an empty Customer class. On the other hand, it feels like we are touching on too many things in this first little test. The second solution I thought about would be to mock the Customer and thereby create looser coupling between the Aggregates, at least in this test. I certainly do like the idea of mocks, but it feels a little bit like overkill here. The third solution could be to create an interface called ICustomer or something similar, or let Customer be an interface and then create a stub implementation of the interface. Again, that feels like overkill right now, and what would the purpose be when moving forward? To be able to swap ICustomer implementations? Is that really something I expect? The fourth solution could be to skip the Customer parameter. I could do that temporarily in order to get going and then change the test when I add the Customer parameter later on. Could it have been a premature design decision that TDD revealed in a few seconds? Instead, I could start by letting the user fill in other information about the new order and then ask the user who the customer is. On the other hand, flexibility somewhere usually means complexity somewhere else. For example, assume prices are customer dependent; what price should be used when I add OrderLines to the Order without a defined Customer? What should happen later? The rule was that every Order should have a Customer. The consumer must do something to transform the order to a valid state after the factory has executed. The same goes if you invent some generic interface that all your Domain Model classes should implement, like IEntity, and then use that as a parameter. What we are discussing now is using a sledgehammer to crack a nut. On the other hand, during the time it takes you to write the name of the test, lots of ideas and thoughts and decisions will flee through your mind very fast and semi-consciously. This, of course, depends on your experience with the problem at hand. Now the test compiles, and I get a red bar because I only wrote the signature and only returned null from the CreateOrder method in OrderFactory. The first try could look like this: It could be public, but instead I decide to let it be internal in order to hinder the consumer of the Order class instantiating it except via the OrderFactory. Well, what value does the factory really add? It just adds a bit of complexity. Now I still think there are quite a lot of things it will deal with, such as snapshotting the customer, adding null objects, creating different kinds of orders, and a few other things. Note It feels strange to refactor away from a factory. Also worth pointing out is that I had a CreateOrderLine method in the sketch in Chapter 4, but life is simpler without it. Sure, what we have been discussing all along is how to deal with the customer parameter. So in my opinion an interesting test would be to check that a newly created order has a customer. I think it will be needed not only for my test, but also for some of the requirements. In order to cut down the complexity, I decide on a read-only property like this in the

Order class: Now everything compiles and life is good, and we do get a red bar. The constructor now looks like this, and we get a green bar: What could the semantics around that one be? The order should probably get an initial OrderDate when first created and then a final OrderDate when the order gets the status of Ordered or something like that. I need to add a public OrderDate property. And this time, I let the constructor of the Order set the OrderDate field without adding another parameter to the constructor. I started this section with a diagram describing a subset of the model in Figure The model has evolved slightly, as shown in Figure where I visualize the code. The reason is that I wanted to illustrate that upfront, I just sketch quickly as a help with the thought and communication process. Note I also show the constructor in Figure to give a more detailed view of how the classes work. First, I wanted the call to the constructor of the Order to be moved out to the [SetUp], but then I would have to change the last test slightly regarding the time interval, and that would make it a little less to the point. Moreover, the three tests shown so far are just about the creation, so I like having the calls in the test methods themselves. In my opinion, the next problem naturally is how the OrderRepository should work out. Well, typically an order has some form of unique identification that could be used by humans for identifying one particular order. One common solution is to just assign a new number to each order from an ever-increasing sequence of numbers. In Figure , I show the Order class again after modification and the newly added method in the OrderRepository class. Do we have to give the Order its identity when it is created? I also strongly dislike the coupling between persistence and business rules. So as usual we can then write a very simple test like this: This way we dealt with OrderNumber from the constructor perspective. If the OrderNumber property is read-only, however, how can we give it the value when using the OrderRepository for finding and reconstituting an old Order? GetOrder theOrderNumber ; Assert. I get a red bar. I could go ahead and implement a new method in the OrderRepository that is only there for supporting other tests, but I think this is a good sign of my needing to write another test instead. I need to test saving Orders, or at least make the Repository aware of Orders. Then I write another test that looks like this: I need to add the AddOrder method to the OrderRepository. And as usual, I just add the signature, but this is not enough to get a red bar. As a matter of fact, there is no test code at all in the CanAddOrder method. The implementation of this method is far from even being started yet. Instead, I take a step back and add a private IList to the Repository for holding on to the orders. Instead I use another assertion, not a xUnit one, but an assertion from the Diagnostics namespace for checking what I think should be checked. What the Assert does is check that the statement is true. If not, the developer will be notified. The AddOrder method could look like this: You should think twice before using the ordinary Diagnostics assertions. I discussed this some more in my earlier book [Nilsson NED]. What could be the pre-conditions that the AddOrder requires? No, I dislike seeing that as an error. But the method would throw an exception if it is. Assert , which will just throw an exception if it receives false as the parameter. Compile, test, and red bar. Add order ; MyTrace. It could look like this: Do you see it? Yep, how can we get the faked OrderNumber into the order instance? As a matter of fact, this is a generic problem. It can be expressed like this: How can we from the outside such as in a Repository set values in instances that are being reconstituted from persistence? Reconstituting an Entity from Persistence: How to Set Values from the Outside I mentioned the generic problem of setting values in an instance that is being recreated by reading it back from the database. If the consumer sets a new OrderDate, there might need to be some checks kicking in. There are several possible ways in which to deal with this problem.

REFINING THE DOMAIN MODEL pdf

8: Advanced Topic â€“ Domain Modeling â€“ Scaled Agile Framework

(Close Table) Projects: Start iteration 3 at end of class []Chapter 31 Refining the Domain Model A long chapter with lots of notation which you need to master.

The goal is to select a model that provides efficient replication of information with minimal impact on available network bandwidth. Number of users in your organization. If your organization includes a large number of users, deploying more than one domain enables you to partition your data and gives you more control over the amount of replication traffic that will pass through a given network connection. This makes it possible for you to control where data is replicated and reduce the load created by replication traffic on slow links in your network. The simplest domain design is a single domain. In a single domain design, all information is replicated to all of the domain controllers. If necessary, however, you can deploy additional regional domains. This might occur if portions of the network infrastructure are connected by slow links, and the forest owner wants to be sure that replication traffic does not exceed the capacity that has been allocated to AD DS. It is best to minimize the number of domains that you deploy in your forest. This reduces the overall complexity of the deployment and, as a result, reduces total cost of ownership. The following table lists the administrative costs associated with adding regional domains. Cost Implications Management of multiple service administrator groups Each domain has its own service administrator groups that need to be managed independently. The membership of these service administrator groups must be carefully controlled. Maintaining consistency among Group Policy settings that are common to multiple domains Group Policy settings that need to be applied forest-wide must be applied separately to each individual domain in the forest. Maintaining consistency among access control and auditing settings that are common to multiple domains Access control and auditing settings that need to be applied across the forest must be applied separately to each individual domain in the forest. Increased likelihood of objects moving between domains The greater the number of domains, the greater the likelihood that users will need to move from one domain to another. This move can potentially impact end users. Note Windows Server fine-grained password and account lockout policies can also impact the domain design model that you select. Before this release of Windows Server , you could apply only one password and account lockout policy, which is specified in the domain Default Domain Policy, to all users in the domain. As a result, if you wanted different password and account lockout settings for different sets of users, you had to either create a password filter or deploy multiple domains. You can now use fine-grained password policies to specify multiple password policies and to apply different password restrictions and account lockout policies to different sets of users within a single domain. Single domain model A single domain model is the easiest to administer and the least expensive to maintain. It consists of a forest that contains a single domain. This domain is the forest root domain, and it contains all of the user and group accounts in the forest. A single domain forest model reduces administrative complexity by providing the following advantages: Any domain controller can authenticate any user in the forest. All domain controllers can be global catalogs, so you do not need to plan for global catalog server placement. In a single domain forest, all directory data is replicated to all geographic locations that host domain controllers. While this model is the easiest to manage, it also creates the most replication traffic of the two domain models. Partitioning the directory into multiple domains limits the replication of objects to specific geographic regions but results in more administrative overhead. Regional domain model All object data within a domain is replicated to all domain controllers in that domain. For this reason, if your forest includes a large number of users that are distributed across different geographic locations connected by a wide area network WAN , you might need to deploy regional domains to reduce replication traffic over the WAN links. Geographically based regional domains can be organized according to network WAN connectivity. The regional domain model enables you to maintain a stable environment over time. Base the regions used to define domains in your model on stable elements, such as continental boundaries. Domains based on other factors, such as groups within the organization, can change frequently and might require you to restructure your environment. The regional domain model consists of a forest root domain and one or more regional domains. Creating a regional domain

model design involves identifying what domain is the forest root domain and determining the number of additional domains that are required to meet your replication requirements. If your organization includes groups that require data isolation or service isolation from other groups in the organization, create a separate forest for these groups. Domains do not provide data isolation or service isolation.

REFINING THE DOMAIN MODEL pdf

9: Domain-Driven Design â€” Flow Framework x-dev documentation

Refining the Domain Model into a Software Architecture Finding a suitable application partitioning depends on framing answers to several key questions and.

The basic idea is that the design of your software should directly reflect the Domain and the Domain-Logic of the business- problem you want to solve with your application. That helps understanding the problem as well as the implementation and increases maintainability of the software. The Domain Driven Design approach introduces common principles and patterns that should be used when modeling your Domain. There are the "building blocks" that should be used to build your domain model and principles that helps to have a nice "supple design" in your implementation. This article tries to introduce some of the concepts shortly and should inspire to read more. But reading this article still requires some minutes - so take your time: But the domain experts know their domain and the developer has to understand it. The domain experts know rules of their business process but often they are not aware using them because it is natural for them. Therefore "knowledge crunching" is required to identify a proper domain model DM. The domain model offers a simplified, abstract view of the problem. It can have several illustrations: Defining the domain model is a cyclic process of refining the domain model. The DM grows and changes because knowledge grows during implementation and analysis! Of course the domain model must be useable for implementation. Ubiquitous Language In order to understand the problem area domain of your application a common language should be used to describe the problem. The goal is that this language and the common vocabulary in it can be understood by the developers and the domain experts e. During the analysis the ubiquitous language should be used to describe and discuss problems and requirements. If there are new requirements it means new words enter the common language That helps refining the model. Use of UML class diagrams to sketch the main classes. Use structure packages and subpackages. Use freetext to explain constrains and relations. Present a skeleton of ideas Layered Architecture The basic principle for software that is build with domain driven design is to use a layered architecture. Where the heart of the software is the domain model. Basic principles of layered architecture is that a layer never knows something about the layers above. They are connected with associations relations. Common pattern like "repositories" and "factories" helps completing the model. Associations between elements Avoid many association and use only a minimum of relations because they decrease maintainability! For example "address", "color" Ideally the only have getter methods and their attributes are set during construction constructor method. A good example is "color" that can be implemented as value object. That means it is an object that represents a certain color. Since value objects are immutable it is not allowed to change the color object. So if you need to get a color that is lighter than another color: They define an easy interface to use in client objects. Aggregates helps to limit dependencies. An aggregate is a group of objects that belong together a group of individual objects that represents a unit. Each aggregate has a aggregate root. The client-objects only "talk" to the aggregate root. Invariants and rules have to be maintained. The aggregate root normaly takes care of invariants. A factory hides logic for building objects - this is especially relevant for aggregates! A factory can be implemented as a seperate object or also as a embedded factory method. Instead of a seperate factory object - a factory method can be located in existing elements: Constructors Factory functionality should only be located in the constructor if creation is simple and if there are good reasons. A constructor should always be atomic never call other constructors. Details can be added later if they are not required by an invariant. The factory knows from where to get the identifier for an entity. Value factory Since value objects are immutable a factory or factory method for a value object takes the full description as parameters. Get references of entities and objects: To do anything with an object you need to have a reference to it: So how to get it? There are several options: For example that have to be used for inner aggregates of course. But there are some entities that need to be accessible through search based on attributes. Repositories For each object where you need global access create a repository object that can provide the illusion of an in memory collection of all objects of that type. Setup access through a well knows global interface. Decouple client from technical storage Performance tuning possible Communicate design decisions related to data

access keep model focus dummy implementation for unit testing is possible. Repositories can hide the OR mapping. If the repository is responsible for retrieving a certain entity that is build by a factory, the repository normally calls the "reconstitution" method of the factory. Bigger picture This diagram taken from the book is an overview showing the parts of domain driven design. Beside the building blocks Eric Evans also pointed out principles that should help to have a supple design in the implementation. One of the main goal of a supple design is that you have code where the developers love to work with - because it is understandable, logic, maintainable and extendable. This are some of the the principles: As much as possible should be query no change to the state of the model. Commands should be clear and simple. If there are unavoidable sideeffects they should be made explicit through assertions. Observe the axes of change and stability through successive refactorings and look for the underlying conceptual contours that explain these shearing patterns. Align the model with the consistent aspects of the domain that make it a viable area of knowledge in the first place. You should use this where possible because no new dependencies are introduced when using this method. DDD supports the agile paradigm and ideally goes hand and hand with refactoring, test driven development and an agile project management like scrum.

REFINING THE DOMAIN MODEL pdf

Real estate office desk book for appraising residential property Multimodal Imaging and Hybrid Scanners List of officers and members of the New York Society Prologue : Land of the dead Freshwater use trends in Maryland, 1985-2000 Political elite in tribal society Dictionary of roman coins Exceptional children an introduction to special education 9th edition Constitutional case of the millenium The Captains Honor Programmable logic device handbook Monotype castor 106 Time. Confinement of general terms Lisa renee jones denial Planning and Designing Clinical Research Mavis gallant paris stories Chinese journal of chemistry The police, a policy paper The pareto principle and drafting mistakes Apostles of Denial Understanding quantum field theory Manufacturing processes for design professionals by rob thompson Analyzing variance with ANOVA Current preparations for World War III-sections A. B. Where the World Does Not Follow Selected Stories of Bret Harte Return to Liverpool and Trip to Europe Proceedings of the Judiciary Committee of the Senate in the matter of the investigation demanded by Senat What makes gravity? Preaching, practice and participation Nicene And Post Nicene Christianity Bangla electrical engineering book Progress of the British North American exploring expedition under the command of Capt. John Palliser, F.R Ane meruellous discours vpon the lyfe, deides, and behauiours of Katherine de Medicis Biology mcq book Stretching the welfare check David Zucchino Developments concerning national emergency with respect to Iran Cancellation of dollar amounts of discretionary budget authority Landslides Under Static and Dynamic Conditions: Analysis, Monitoring, and Mitigation Never too old to knit