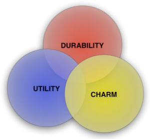


Class 8 SSW565

Gregg Vesonder
Stevens Institute of Technology

Roadmap

- Logbook
  - me
  - Volunteer
- The Arch Discovery
- Part 2 in Evans
- Readings next class, chapters 8 thru 13 in Evans, <u>Domain Driven Design</u>

Key Dates

- Thursday class June 25th
- June 29th logbooks due
- Final July 20th – take home due July 23rd

Logbook

- No coffee!
- Your turn

Written Design Documents

- Even though Agile and XP are light on documents, they have their place in “rational” Agile methodology
- Issue is that once a document takes on a persistent form (is baselined) it is difficult to keep it current with the project
  - Documents should complement code and speech
  - Relying on code as the sole design document can be overwhelming
  - Documents can provide insight on the large scale structure of the code and focus attention on core elements -- it should be light on specifics, that is what code does well and also hard to keep current. Documents should not try to do what code does well

View on Design

Design View	Description	Attributes	User Roles
Decomposition	Decomp of system into modules	Identification, type, purpose, function, subcomponents	Project manager
Dependencies	Relations between modules and resources	Identification, type, purpose, dependencies, resources	Configuration manager, maintenance programmer, integration tester
Interface	How to use modules	Identification, function, interfaces	Designer, Integration tester
Detail	Internal details of modules	Identification, computation, data	Module tester, programmer

More on the Model

- Documents should work for a living and stay current -- it can be as straightforward as a set of informal sketches
  - Documentation must be involved in project activities
  - If a document is not used by the team - toss it
  - Of course it must use the Ubi language
- Again code must work with the documentation and the UML
  - It takes a lot to write code that doesn't just do the right thing but also says the right thing! Executable bedrock
- Finally - one model should support implementation, design and team communication -- there also can be an explanatory model for other stakeholders

Binding Model and Implementation

- The model is more than analysis of the problem it is the foundation of design
  - Can only be accomplished if you tightly relate code to the model -- in fact make it part of the model
- Analysis models are the result of analyzing the business domain to organize its concepts without consideration to the role it plays in software - a tool for understanding
- This analysis model is necessarily incomplete because crucial insights and discoveries always emerge during design and implementation
- At the same time software that lacks a concept as the foundation of the design is a mechanism doing useful things without explaining its actions, leading to all sorts of issues later in the life cycle

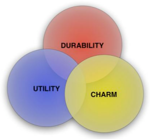
OO

- OO analysis and design
- European : American :: modeling : reuse
- Class represents what an object is about & relationships
- Relationships: generalization - specialization
- Object is, state + behavior
- What are objects, from requirements: nouns, roles, … The behavior of the objects are verbs
- Difficulty, users think in tasks not objects, case studies bridge this
- CRC - informal
- The UML
- Design Patterns: MVC & wrappers (legacy)
- Metrics really matter
- The great developer and application divide - 90s

OO Analysis and Design

- OO Analysis = Requirements analysis + Domain class selection
  - Product = Complete requirements document + domain class model + basic sequence diagrams
  - Domain classes obtained via use cases -> sequence diagrams and brainstorming/editing process
  - Use domain classes to organize requirements
- OO Design = all other activities except coding
  - Product = complete detailed design ready for coding

Eric Braude, Software Design, John Wiley, 2004.

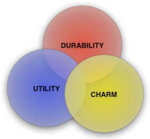


OO Analysis and Design

- Traditional techniques focus on functions of the system
- OO focuses on identifying and interrelating the objects that play a role in the system
- Convergence to UML, Unified Modeling Language, Booch, Jacobson and Rumbaugh
- Heuristic thoughts- keep objects simple and each method should send messages to objects of a very limited set of classes (more when we explore OO metrics)

“Schools” of OO

- European school, influenced by the Scandinavian school of Programming, OO analysis and design is modeling real world objects both animate and inanimate
- *American school,* OO focuses on data abstraction and component reuse - identifying reusable components and building an inheritance hierarchy.
  - “What matters is not how closely we model today’s reality but how extensible and reusable our software is”



OO viewpoints

- Modeling (European) viewpoint - conceptual model of some part of a real or imaginary world.
  - Each object has identity, is unique
  - Objects have substance, properties that hold and can be discovered
  - Objects are implementations of abstract data types
    - Mutable state, variables of abstract data type
    - Operators to modify or inspect the state
      - Only way to access object
      - Interface to object
  - Object = identity + variables + operators or
  - Object = identity + state + behavior

OO Viewpoints - 2

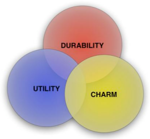
- Philosophical view - objects as existential abstractions, the unifying notion underlying all computation
  - Beginning and end to objects
  - Eternal objects, e.g., integers
    - Not instantiated, cannot be changed
- Software Engineering view - data abstractions encapsulating data and operations
  - Object based languages encapsulates abstract data types in modules whereas
  - Object oriented also includes inheritance

OO Viewpoints - 3

- Implementation viewpoint
  - Contiguous structure in memory, a record of data and code elements
- Formal viewpoint
  - Object viewed as a state machine with a finite set of states and a finite set of state functions. State functions map old states and inputs to new states and inputs
- While modeling conceptual viewpoint is stressed
- Tensions between a problem oriented (analysis) vs. solution oriented viewpoint (design)

Model Driven Design

- Model driven design uses a single model both as an analysis and design model
  - Test the model by implementing a bit of it, then revisit the model and tweak so that it is easier to implement
  - cyclic developing the code and the model and ubi language
- The OO methodology strongly supports such a process
- Development therefore becomes an iterative process of refining the model, the design and the code as a single activity



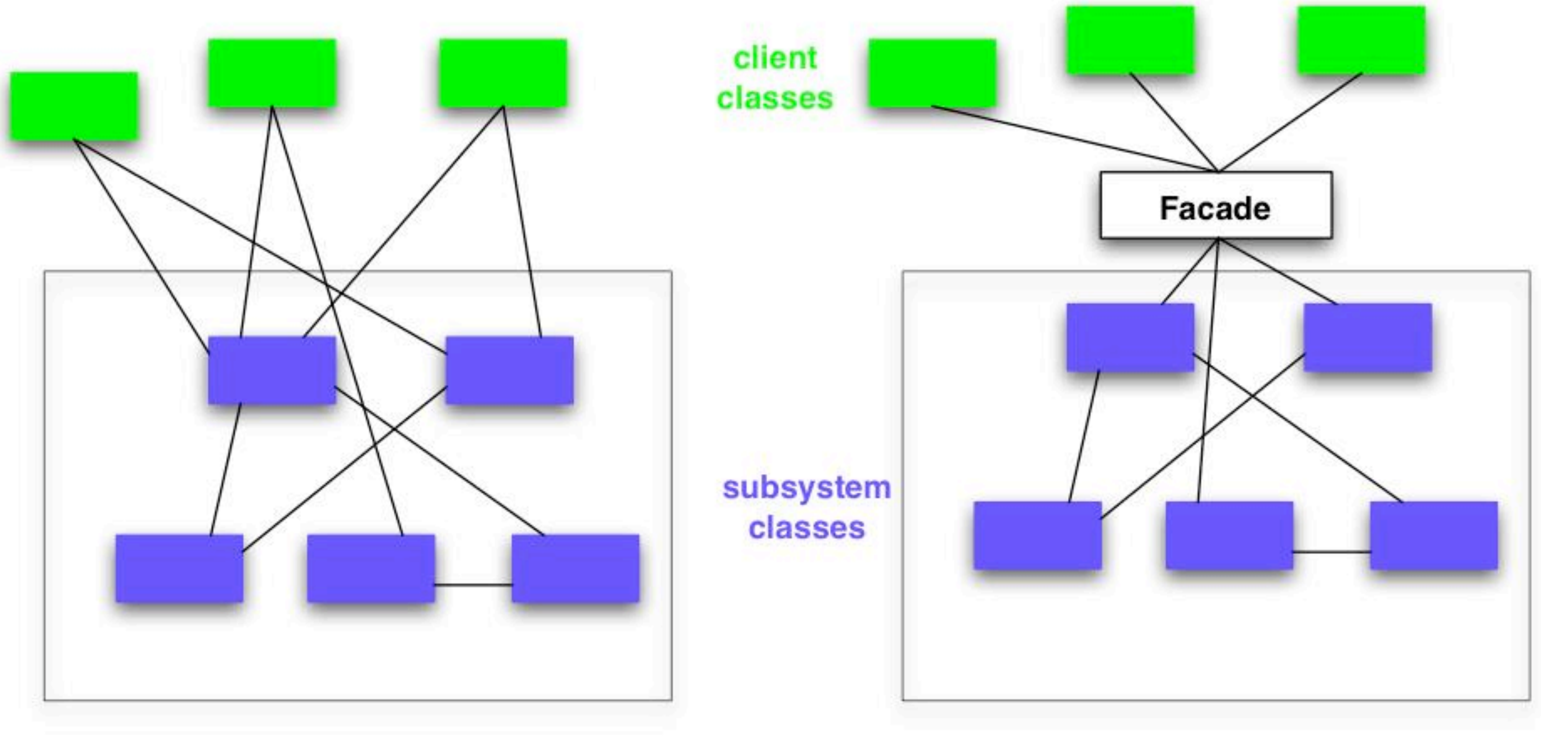
Object Design

- Real break through of OO design comes when the code expresses the concepts of the model
- There is no modeling paradigm that supports a procedural language such as C
- In one of the examples, the façade pattern is mentioned

Facade

- **Pattern Name** - façade (I sometimes call it veneer)
- **Problem** - need a simpler interface, a unified interface to a subsystem. Example is a single call to compiler versus calling all of its necessary subtasks, scanner, parser, … Note that the classes still exist so that other clients can use them if need be. Also when there is a need for a novice versus expert interface.
- **Solution** - A single class façade that is interface to all the subsystem classes
- **Consequences** - makes it easier for client and provides weak coupling so you can change the elements of the subsystem that are being used. At the same time the subsystem components are still there to be used.

Facade



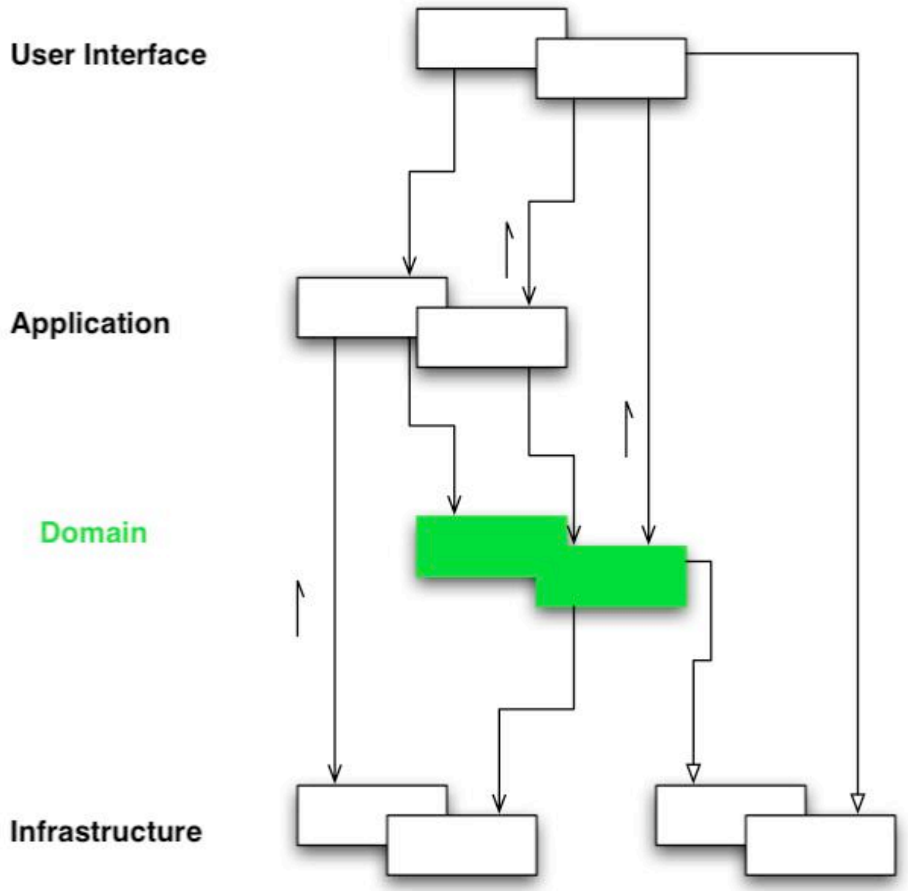
Hands on Modelers

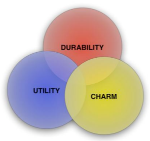
- Letting the bones show - why does the model matter to users?
- In this method software development is ALL design
  - Code adds the details and is part of the design, you cannot design at a certain level without coding. Why use pseudo-code when you can use code!
- If developers do not have responsibility for the model (or don't understand it), then the model has nothing to do with the software
- Conversely when a modeler loses touch with the code, the modeler loses touch with the constraints of the implementation

Part 2 - Building Blocks

- Layered architecture is key
- Creating programs that can handle complex tasks requires “separation of concerns” permitting concentration on different aspects of the design in isolation
  - Each layer specializes on a particular aspect of the program
  - Domain layer is the key to model driven design
  - The domain layer, not the application layer is responsible for fundamental business rules

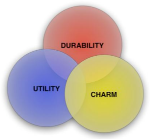
Layered Architecture





SERVICES -> Layers

Application	Funds Transfer Application Service
	Digests input (XML request)
	Sends message to domain service for fulfillment
	Listens for confirmation
	Decides to send notification using infrastructure service
Domain	Funds Transfer Domain Service
	Interacts with necessary account and Ledger objects, debiting and crediting
	Supplies confirmation of the result
Infrastructure	Send notification service
	Sends email, letters and other communications as directed by application



Relating the Layers

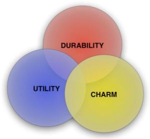
- Of course there are architecture patterns for relating the layers (callbacks and observers)
- UI to application uses the Model-View-Controller pattern
- Infrastructure layer objects are typically offered as services
- Application knows what, not how, e.g., knows when to send a message but not how to send it
- The Domain layer is where the model lives

Observer

- Pattern Name - observer, aka dependents, publish-subscribe
- Problem - Problem of maintaining consistency between/ among related objects.
- Solution - a subject (object being observed) and one or more observers are implemented and all observers are notified when subject changes state, and then each observer synchronizes with subject. Subject is publisher, observers subscribe to the notifications.

Observer - 2

- Consequences - can vary subjects and observers independently, loosely coupled, support broadcast. As more observers subscribe to a subject, it can be costly changing the subject. Also if it is a simple broadcast (stating simply that a change has occurred but not specifying the change), observers may have significant work finding what has changed.

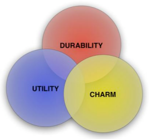


The Smart UI

- When circumstances (simple or small applications) warrant put all the business logic in the UI
- Advantages:
  - High productivity for simple applications
  - Less capable developers can do this (domain modeling requires high class developers as does XP
  - Use the prototype of the UI to debug requirements
  - Applications are decoupled so delivery of small modules can be planned accurately and expansion is easy, just add more applications
  - Relational database works well with this scheme
  - 4GL works best rather than OO language
  - Easy to do maintenance, effects of change are localized

Smart UI Disadvantages

- Integration of Applications difficult, must use DB as the intermediary
- No reuse of behavior, no abstraction of business problem -- business rules have to be duplicated
- Rapid prototypes reach a limit since lack of abstraction limits refactoring
- Complexity overwhelms the Smart UI, there is no path to richer applications and behavior



Model Expressed in Software

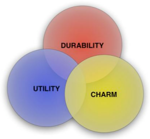
- Three elements express the model: Entities, Value Objects and Services
  - Entity-something that is tracked through different states and even different implementations
  - Value Object - an attribute that describes the state of something else
  - Services - some aspects of the domain are expressed as actions or operations, something that is done for a client by request, prevalent especially in the technical layers of the model
- Modules are part of the model and they should represent concepts in the domain -- locus of high cohesion, low coupling

Associations

- Of course these elements are associated and the key in model building is to carefully distill and constrain the model’s associations
- Three ways to maintain control over associations
  - Imposing a traversal direction
  - Adding a qualifier, reducing multiplicity
  - Eliminating associations

Relations between objects

Relationship	**Example**
Specialization/ generalization, isa	Table isa furniture
Whole-part, has	Table has tabletop
Member-of, has	Library has member

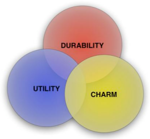


More on relations

- Generalization-specialization can be expressed as a hierarchy
  - Single inheritance, tree
  - Multiple inheritance, directed acyclic graph
  - Define common attributes at a higher level and let descendants inherit the attributes
  - Object hierarchy can be viewed as a type hierarchy, chair and table are subtypes of furniture

More on Relations - 2

- Part of relationship aggregates components into a whole
  - It is a transitive relationship
- Member-of relationship represents the relation of a set and its neighbors
  - It is not transitive



Entities (Reference Objects)

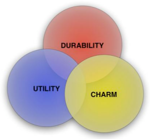
- Many objects are not primarily defined by attributes but by continuity and identity - often across distinct representations
- An object defined primarily by its identity is an ENTITY
  - Have life cycles that can radically change their form and content
  - Their class definitions, responsibilities, attributes should revolve around who they are
- An ENTITY is anything that has continuity through a life cycle and distinctions independent of attributes, e.g., person, car, city, lottery ticket
- Transactions have identity, the amount attributes of the transaction do not have identity

Defining an ENTITY

- Keep class definitions simple and focus on life cycle continuity and identity.
- Attach a symbol that is guaranteed unique
  - May come from external world (social security number) or the system -- gensym
  - Defines what it means to be the same
- Object definition should be based on things that identify it or things that help find or match it
  - Add only behavior or attributes related to these.
  - Customer ID is the identifier but phone number or address can be used to find it.

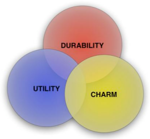
Designing the Identity Operation

- Identifying attribute must be unique even if distributed or if archived (easier if external, but issues of privacy)
- Defining identity is not technical it requires understanding the domain
  - Sometimes certain data attributes or combinations of attributes can be guaranteed unique
  - Unique id must be designated immutable
    - Whatever happens it remains
    - System can generate a unique id but challenging in distributed systems - lots of techniques, e.g, provide each system with number ranges
  - If it is automatically generated user may never see the ID only the system uses it, except in cases such as tracking number for UPS, ..



VALUE Objects

- You could assign identity to all domain objects (and it has been done) but that is a lot of tracking
  - Muddles the model, all objects are the same -- you want it to be much more enlightening
- VALUE objects, objects that describe things. Depending on the Domain, some things can be ENTITIES or VALUE objects, e.g., address can be an ENTITY for the post office and a VALUE object for a mail order company (persons at address are ENTITY)
- VALUE objects are instantiated for what they are, not who or which they are
- Example of VALUE objects are any assemblage of things such as color or in some cases value objects are made up of other value objects (the window example in the book, dimensions, wood type, …)

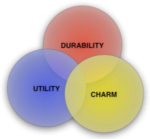


More VALUE Objects

- They can reference ENTITIES, e.g., a route references highways and streets
- Often passed as parameters, often transient or needed as attributes for ENTITIES - a person's name is a VALUE
- In general VALUE objects should be immutable
- Attributes that constitute a VALUE object should be treated as a whole, e.g., address = street +city + postal code

Designing VALUE Objects

- Name objects, e.g., Ralph, can be shared by all the Ralphs, of course it must be immutable since all Ralph's depend on it
  - Therefore to change your name to Roy, you change the pointer from Ralph to Roy VALUE name object, not change the Ralph object
  - If you need to pass something in a method that may change, pass a copy
  - This is an example of the Flyweight pattern have one object point to it many times - an optimization trick not available for ENTITIES, e.g., there can be dozens of Ralphs

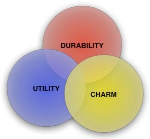


Flyweight

- Pattern Name - flyweight
- Problem - some applications would benefit from creating many objects for maxim flexibility but the resulting overhead would be difficult to manage for the system and the resources.
- Solution - provides a scheme to share objects allowing the benefit of many objects without the cost. The key is differentiating between intrinsic and extrinsic state. The intrinsic state is the information that does not vary by context, e.g. a character code. However its position in the document and font, size, … does vary and are represented in the extrinsic component.

Flyweight - 2

- Solution (cont'd) The extrinsic component provides the context of that particular use of the object. Note this only works if there are core features that can be intrinsic and the bulk of the object definition is extrinsic. Note also that you cannot use object identity since the core object is reused/shared. Client has the burden of maintaining all the extrinsic information
- Consequences - savings are the key and depends on amount of intrinsic state, reduction in total instances by using this scheme, whether extrinsic state is computed or stored, where computed is definitely better (no space is required).



VALUE Objects: Copying vs. Sharing

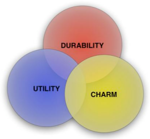
- Economy of copying versus sharing depends on environment - copies can clog a system but sharing can be slow in a distributed system
- Denormalization is the technique of storing multiple copies of the same (immutable) data
- Sharing restricted to:
  - When space saving or limiting object count is crucial
  - When communication overhead is, low
  - When the shared object is strictly immutable

VALUE Objects: When Change is Good

- Factors favoring a mutable VALUE object:
  - Value changes frequently
  - If object creation or deletion is expensive
  - If replacement will disturb clustering in a distributed environment
  - If there is not much sharing of values or it is avoided to improve clustering
- Of course if a VALUE’s implementation is mutable it must not be shared
- Strive to design VALUE objects as immutable

SERVICES

- Often a capability of a model is neither an ENTITY or a VALUE and many of these capabilities are activities or actions - since we have an OO methodology, should be expressed in objects -- SERVICES
- Usually SERVICES have no state of their own and their meaning is tied to the operation they host
  - They are defined solely in terms of what they can do for the client
  - Activity “verb” rather than a “noun”
  - Should be declared in the model as a SERVICE and be in the UBI language



Characteristics of a Good SERVICE

- Operation that relates to a domain concept and is not a natural part of a VALUE or ENTITY object
- Interface is defined in terms of other elements of the domain model (someone requests it)
- Operation is stateless, any client can use any instance of the SERVICE
  - However services can change global information

SERVICES and the Domain Layer

- Many SERVICES naturally fall in the infrastructure layer
- Often hard to differentiate application layer SERVICES from domain layer SERVICES
  - The application layer is often responsible for ordering the service (application is when/what, not how)
- Remember domain layer is responsible for domain knowledge, e.g., business rules

SERVICES -> Layers

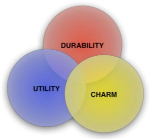
Application	Funds Transfer Application Service
	Digests input (XML request)
	Sends message to domain service for fulfillment
	Listens for confirmation
	Decides to send notification using infrastructure service
Domain	Funds Transfer Domain Service
	Interacts with necessary account and Ledger objects, debiting and crediting
	Supplies confirmation of the result
Infrastructure	Send notification service
	Sends email, letters and other communications as directed by application

Last Words On SERVICES

- Helps to keep other objects from being too detailed -- all the code would be replicated in each of the objects
- Singleton pattern can provide access to services
- J2EE & CORBA are not necessary to provide services and sometimes can be overkill

Singleton

- Pattern Name - Singleton
- Problem - situations where a class can only have one instance, for example a thread pool manager or print spooler.
- Solution - devise a class that is responsible for keeping tracking and providing access to one and only one instance. This instance needs to be extensible (be modified) through subclassing.
- Consequences - access is controlled to instance, it avoids using global variables, subclass permits evolution. You can modify to provide a variable number of instances. More general than language provided features (e.g., static member functions in C++).



Modules (Packages)

- Modules tell the domain story at a higher level
- Modules are often not treated as aspects of the model
  - Modules often reflect other concerns such as that of the technical architecture or partitioning work
- Should exhibit high cohesion, low coupling
  - Think of concepts being divided into modules, not code
  - Especially important at higher levels of the model
  - Modules are human communication mechanisms equivalent to chapters in a book - name should convey its meaning

Other Aspects of Modules

- “agile” Modules -- as coupling increases you may have to refactor or reorganize modules
- Beware of technically driven module scheme
  - It can run at cross purposes to the model separating objects into layers for technical rather than conceptual reasons
    - Can go against cohesion and coupling rules too

Non Objects in an Object World

- OO predominates in modeling because of balance of simplicity and sophistication, it is also widespread and has strong support
- However model paradigms have been conceived to address certain ways folks think about domains, e.g., rules, scientific calculations, work flow, relational databases
- Mixing is doable, but with care, e.g., in using rules it is necessary to have tight, clear relationships between rules and objects

Rules for Mixing Techniques

- There are always several ways to do something, objects may still be fine
- Lean on the UBI language to make it clear
- Don't get hung up on UML - rules may be best expressed in text
- Be skeptical, is the rules engine pulling its weight? Is it really necessary or could you cast the rules as objects but less neatly?

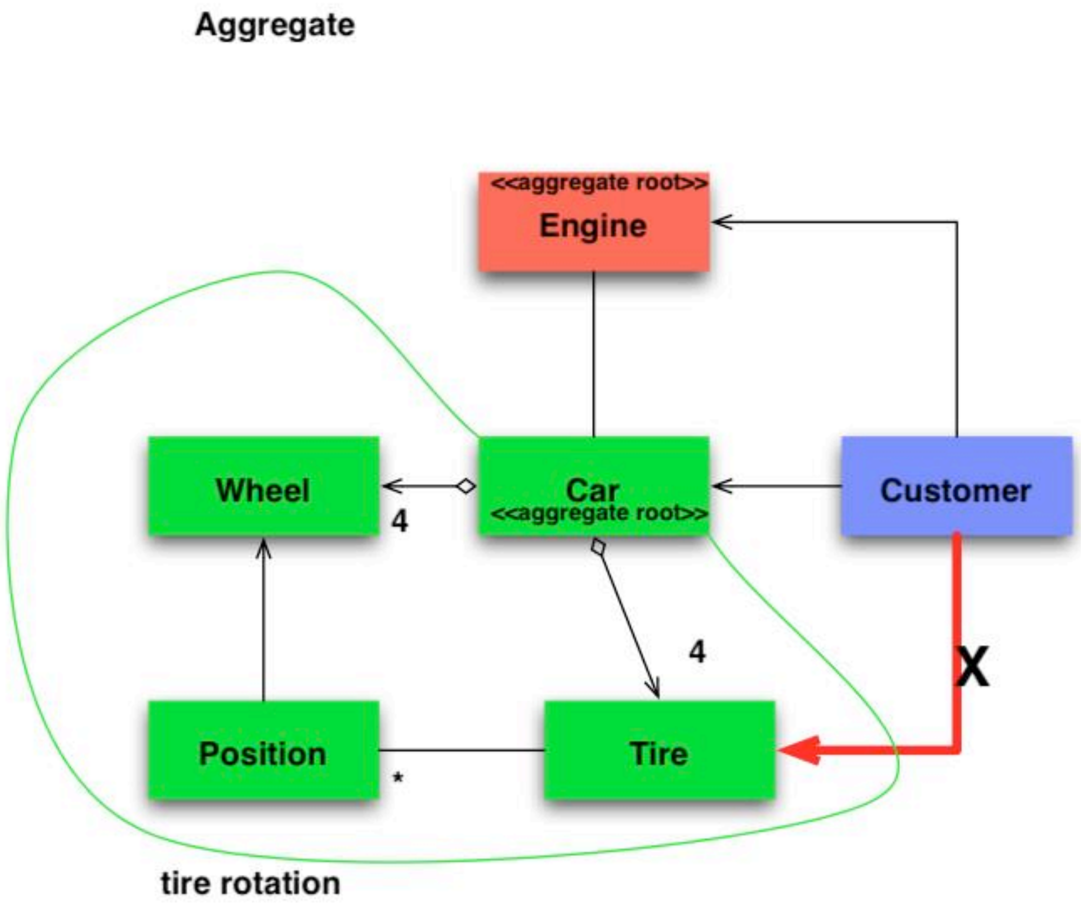
Life Cycle of a Domain Object

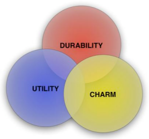
- ENTITY objects are born, go through states and die
- Some objects persist for a long time and they are challenging
  - Maintaining objects integrity throughout the life cycle
  - Preventing model from being swamped by the complexities of managing the life cycle
- Address these issues in 3 ways:
  - Aggregates - defining clear ownership and boundaries for objects
  - Factories - established mechanisms to create and reconstitute objects
  - Repositories - address persistence and middle and end of life cycle.

Aggregates

- An abstraction for encapsulating objects within the model, a cluster of associated objects that we treat as a unit for purposes of data changes
  - Prevents simultaneous changes to interdependent objects
  - Root is only member of objects that outside objects hold reference to, however objects within may reference each other
  - Root entities have global identity, entities inside the boundary have local identity
  - Only aggregate roots can be obtained by database query - everything else by traversal of associations
  - A delete operation must remove everything within the aggregate boundary at once
  - When a change within an aggregate boundary is committed all invariants must be satisfied
  - Factories and repositories operate on aggregates

AGGREGATES


Aggregate
<<aggregate root>>
Engine
Wheel
4
Car
<<aggregate root>>
Customer
4
X
Position
*
Tire
tire rotation



Factories

- When object creation or aggregate creation becomes too complicated or reveals too much internal structure, Factories provide encapsulation
- Object creation and assembly are a necessity, they usually do not provide meaning in the domain - e.g., open a bank account requires adding elements that are not part of the model (beyond ENTITIES, VALUE objects and SERVICES) but still are essential to the operation of the domain layer.
- Factory is a pattern with the responsibility of creating other objects, creating entire aggregate as a unit enforcing their invariants

Abstract Factory

- Pattern Name - abstract factory, aka kit
- Problem - There is a model of a generic product and a specification of all the necessary members (parts or products if it is a product family) but this generic product has product families or there is a need to separate the specification from the actual implementation. The client initiating the process stays independent of the concrete implementation.
- Solution - define abstract factory with abstract products that are needs of the client. Concrete versions of factory are accessed by client through abstract factory. Abstract products define what concrete classes must be developed for the concrete factory to meet the needs of the client.

Abstract Factory-2

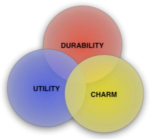
- Consequences - ability to develop code <u>once</u> that relies on abstract factory with knowledge that same set of products, as defined by abstract products, will be created for all concrete instances but client code is isolated from these details. Note that client calls only one thing, the abstract factory, so the concrete factory can be switched. It also enforces the fact that only this combination of product objects works together --- important in look and feel. The bummer is that if you want to add a new product to the set you have to change ALL versions of concrete factory since abstract factory defines what are the set of products for any concrete factory derived from it. Therefore if you want to add a product all concrete factories have to be changed (by adding a new product). Why? Because when the client calls the abstract factory it expects a certain, consistent set of products, so whatever concrete factory, abstract factory points to better have those products.

Builder

- Pattern Name - Builder
- Problem - Separate construction from representation.
- Solution - Builder pattern constructs the product step by step under the Director's control. Focuses on building a complex object step by step.
- Consequences - vary a product's internal representation, better control over the construction process and therefore the internal structure of the product.

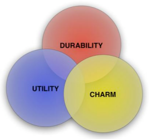
Factory Method

- Pattern Name - Factory Method, aka Virtual Constructor
- Problem - Do not create a family of products (as in Abstract Factory), but a product. Define what an object needs to do (its interface) but separate what it needs to do from its implementation. One variant can specify default implementations for the product.
- Solution - create abstract factory methods that specify what the creator of an object should define (and in some cases define a concrete default implementation). For instance the abstract factory method for door must also state that it needs two places to connect.



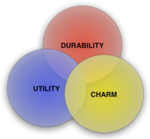
Factory Method -2

- Consequences - sometimes subclassing the creator class just adds another level of indirection w/o adding anything to the design. But using factory method can make explicit connections between parallel class hierarchies. For example if a class of object created consistently needs a class of manipulator that relationship is indicated in the abstract factory method since it states that that class of manipulators is necessary and there likely to be a one to one correspondence between the object class in its hierarchy and the manipulator class in its hierarchy.



Basic Requirement for a Good Factory

- Each creation method is ATOMIC and enforces all invariants of the created object in an aggregate
  - There may be some optional elements to be added later
  - For immutable VALUE objects all attributes are initialized to their final state (tire size)



Factory not a Good Idea

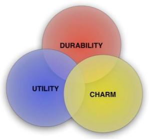
- When the class is the type
- The client cares about implementation, perhaps as a way of choosing the strategy
- All attributes of the object are available to the client, so no object creation gets nested inside the constructor
- When construction is simple
- Public constructor must follow all rules of factory, satisfying all invariants
- Constructors should be dead simple, else use factories (don't call constructors within constructors!)

Designing Interface to the Factory

- When designing the method signature:
  - Each operation must be atomic, you have to pass in everything you need to make a complete product in a single interaction
  - Factory will be coupled to its arguments
- Where does invariant logic go?
  - Factory should delegate invariant checking to the product, the object created
  - Some logic should remain in the factory:
    - Aggregate logic that spans many objects
    - Checking invariants for immutable objects created - only need to be tested once, why gum up product code

ENTITY and VALUE Object Factories

- VALUE objects are usually immutable so they have to emerge complete
- ENTITY Factories tend to require only essential attributes
  - Details can be added later if not required by the invariant
  - Factory is an ideal place to assign identity
    - Actual identity generation may be done by database or infrastructure routine
    - Factory knows how to accomplish this and where to store result

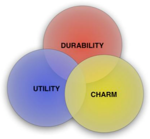


Repositories

- Handles the transition to and from persistent storage
  - Often all objects cannot be kept in memory
  - System is switched on and off
- To do anything to an object you have to hold a reference to it obtained by
  - Creating an object and storing its reference or
  - Traverse an association, starting with object you know and ask for associated object -- key is to get the first object that starts the traversal
  - Execute a query in a database and reconstitute an object

Reconstitution

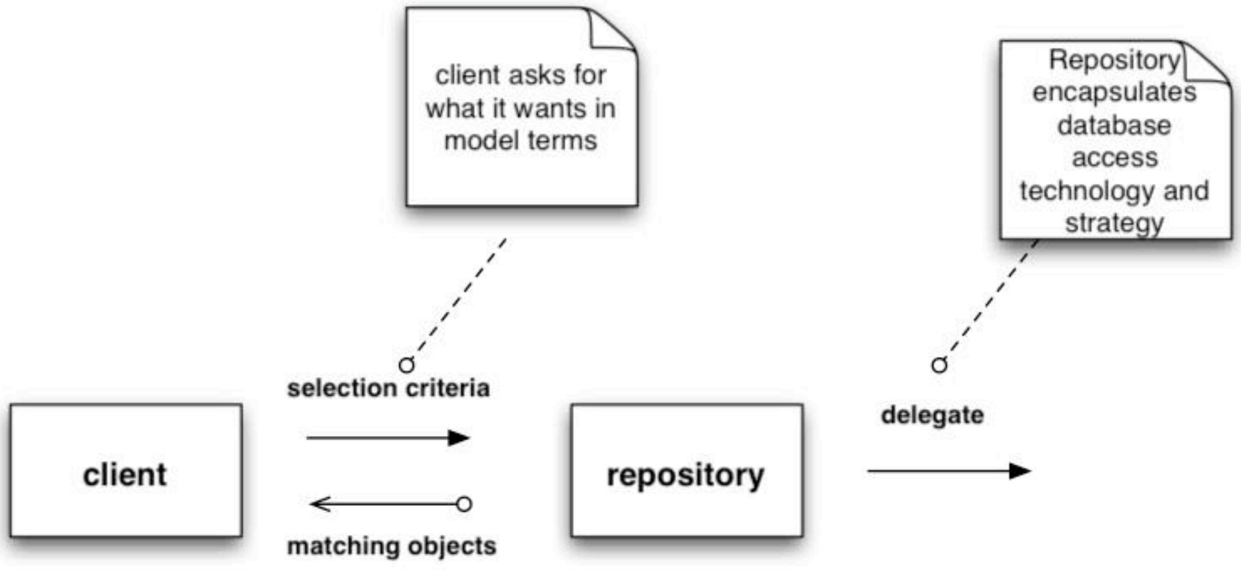
- Retrieval of a stored object is really a subset of creation since we have to create the object anew BUT it is in the middle of the lifecycle for that object
- Reconstitution - the creation of an instance from stored data of that object
- Avoid temptation of assessing the data directly in the database rather than recreating the object
  - Results in an increasing number of domain rules being embedded in query code, making the model increasingly irrelevant
  - Of course performance factors are an issue
- The focus on all of this has been retrieval and not on the model -- the Repository pattern helps establish focus on the model



Repository

- Represents all objects of a certain type as a conceptual set
- Objects are added and removed and the machinery behind the repository adds or deletes them in the database
- Clients request objects from the repository using query methods, selecting objects based on the value of certain attributes provided by the client
- Repository creates an illusion of an in memory collection of that type by providing a simple and hiding all the complexity so developer can get back to model

Repository


client asks for what it wants in model terms
Repository encapsulates database access technology and strategy
selection criteria
client
repository
delegate
matching objects

Query a Repository

- Easiest to build is hard coded queries with specific parameters, e.g.
  - Retrieving an ENTITY by its identity
  - Requesting a collection of objects with a particular attribute value or range
  - Return collection of objects plus some type of secondary attribute (derived) such as obejct count
- More flexible query would be specification based -- client asks what it wants

Implementing a Repository

- Hide all of the inner workings from the client, the client should be the same regardless of selection of storage or change in storage
  - Should not care if it is stored in relational database, flat file, etc.
  - Developer takes advantage of this decoupling to make performance enhancements (developer should understand the inner workings of repository)
  - Enables faking the storage during testing

Repositories and Factories

- Factory's handle the beginning, repositories, everything afterwards
- Repositories reconstitute objects do they are technically factories too, but must always consider from a model basis it is not creation of a new object (although technically it is)
- Can have the repository delegate object creation to a factory.

Designing Objects for Relational Databases

- It is a challenge to map objects to a database
- Three common cases:
  - Database is primarily a repository for objects
  - Database was designed for another system
  - Database designed for this system but has other roles too
- Don't let the data model and object model diverge
- IMPORTANT processes outside the object model should not access the store
- Simplicity is best
  - Row in database table should conform to object
  - Names and elements of objects should correspond to names in the relational table

Other References

- Portland Design Repository
- Gamma, Helm, Johnson & Vlissides Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.