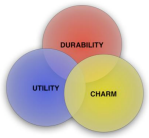
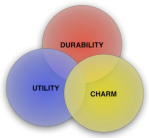


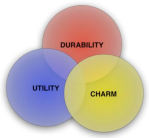
Class 10 SSW565

Gregg Vesonder
Stevens Institute of Technology



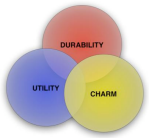
Roadmap

- Logbook
- On Modules
- Part 3 in Evans
- Readings next class, chapters 14 thru 17 in Evans, Domain Driven Design



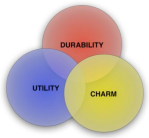
Key Dates

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



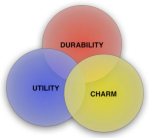
Clarifications

- Part of relationship is transitive – b is part of a, c is part of b therefore c is part of a
- CGI
- Introduction of errors



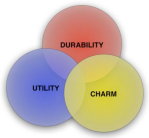
CGI

- CGI – Common Gateway Interface
- A protocol usually employed in scripting languages (e.g., perl) to enhance web pages
- Excellent CGI tutorial: http://www.comp.leeds.ac.uk/Perl/Cgi/start.html



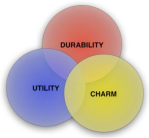
Clarification -Testing

- NOW
- Postponing testing *for too long* is a severe mistake.
- Boehm- errors discovered in the operational phase incur cost 10 to 90 times higher than design phase
  - Over 60% of the errors were introduced during design
  - 2/3's of these not discovered until operations
- Given care you can test requirements specification, architecture and design specification
  - Also prototypes, story boards and even macromedia demos test aspects of the spec, arch and design



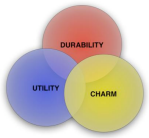
Logbook

- On constraints - Open source and the architects and designers
  - http://www-128.ibm.com/developerworks/
- Your turn



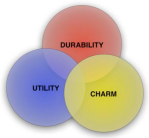
Modularity

- Modules and their interaction
  - Not considered to be an independent (sub-)system, depends on other modules
- Compare designs by considering both a typology for the individual modules and connections between them. Two potential strategies:
  - OO decomposition - set of communicating objects
  - Function oriented pipelining (pipes and filters) receives input data, transforms, outputs
- 2 structural design criteria, cohesion and coupling
- Strive for high cohesion, low coupling



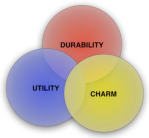
Coupling - Budgen(2003)

FORM	FEATURES	DESIRABILITY
Data Coupling	Modules A & B communicate by parameters or data items	High
Stamp Coupling	Modules A & B make use of a common data type	Moderate
Control Coupling i) Activating ii) Coordinating	i) Transfer control structurally e.g., procedure call ii) A passes control parameter to B - Boolean Flag	i) Necessary ii) Undesirable
Common Environment Coupling	A and B contain references to a shared data area and have knowledge of the data - if data changes …	Undesirable



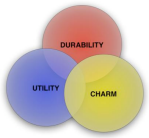
Cohesion - Budgen(2003)

Form	Features	Desirability
Functional	All elements contribute to single domain related task	High
Sequential	Outputs of one step form inputs to other	Quite High
Communicational	All elements operations use same data	Fairly
Procedural	Related by sequential order of operation	Not very
Temporal	Elements related by time, e.g. system initialization	Not very
Logically	Operations are logically similar but not domain related	Not!
Coincidental	Thrown into a bag of code	Not!



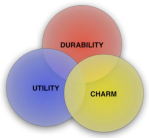
Coupling and Cohesion

- Advantages of low coupling, high cohesion:
  - Communication between developers is easier
  - Correctness proofs are easy to derive and sustain
  - Changes will not affect other modules, lower maintenance costs
  - Reusability is increased
  - Understanding increase
  - Empirical data shows less errors.



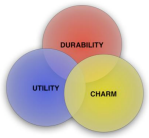
Information Hiding

- Most important aspect
- If a module hides some secret, it does not permeate the module's boundary
- Decreases, coupling, increases cohesion
- But should only hide one secret



Budgen Heuristics

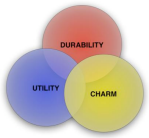
- Anti Information Hiding:
  - Having many copies of state spread around system
  - Complex interfaces (procedures with too many parameters) and complex control structures - see simplicity
  - Needless replication - copy whole record when one field is needed
  - Poorly focused functions - what does it do?



"A paragraph is a collection of statements that accomplish a single task: in literature, it's a series of sentences conveying a single idea; in programming, a series of instructions implementing a single step of an algorithm."

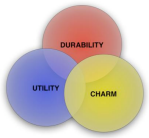
Practical Module Design/Development

- (from PERL, O'Reilly Damian Conway)
- Design Module's interface first, try it on
- Write the test cases - tie to use cases
- Document (standard form?)
- Use revision control system
- Create consistent command line interfaces
- Have an agreed coherent layout style (readability)
- Code in commented "paragraphs"
- Standard error technique
- Add extra test cases after coding before debugging
- Benchmark and compare performance of alternatives

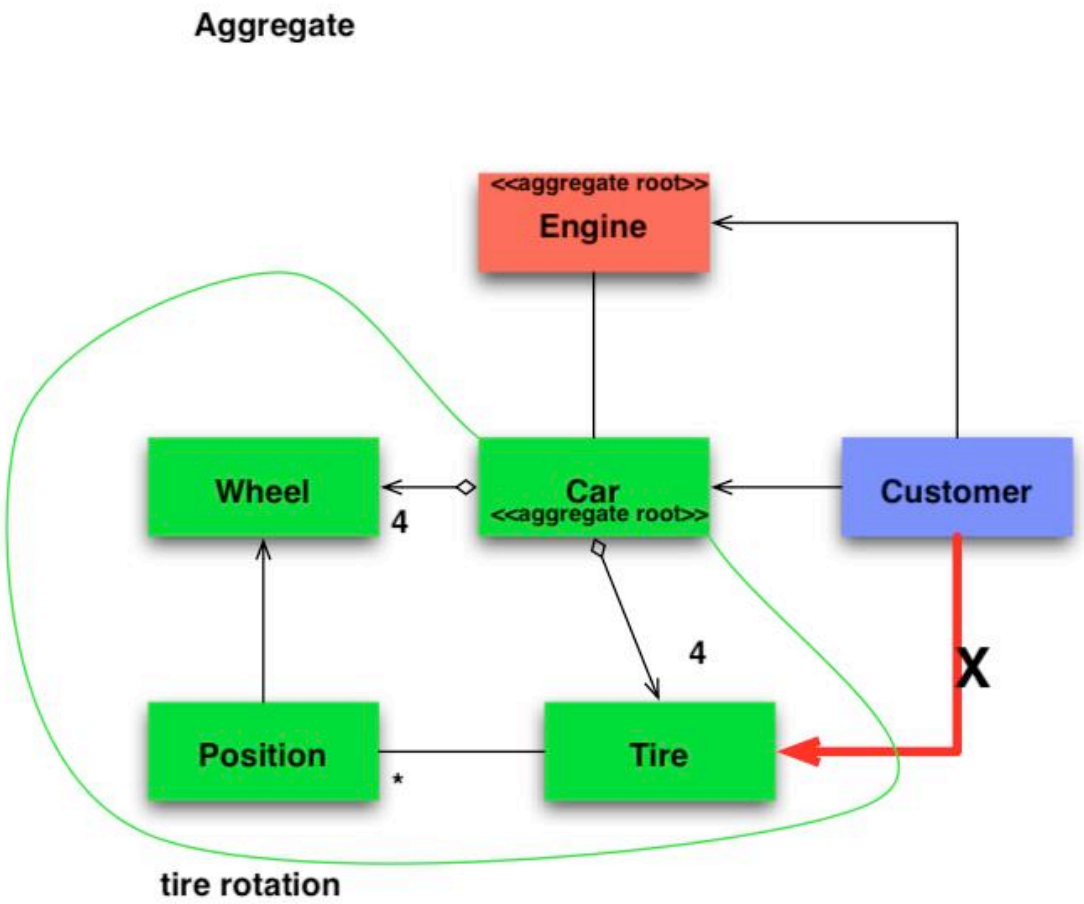


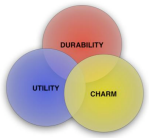
Last Time

- Entities
- Value Objects
- Services
- Aggregates


DURABILITY
UTILITY
CHARM

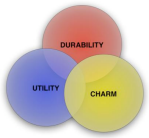
AGGREGATES


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



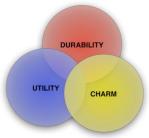
Repositories

- 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



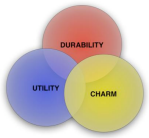
Reconstitution

- 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 accessing 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



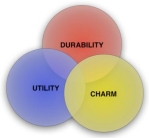
GLOOM

- The concept of a repository is a key component in GLOOM.
- What should and should not be part of the repository?
  - Should - subscription information, identity information
  - Should not - current game state
    - But what if game is to be saved for later continuation?
- Should there be a local repository?



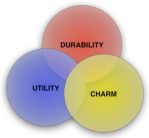
What's a Model

- It is a simplification, a coherent abstraction of the essential points relevant to the present and future tasks in the subject area
- Subject area is also known as domain
- A Domain Model is a rigorously organized, selective abstraction of the Domain Knowledge
- "loosely representing reality to a particular purpose"
  - Utility is the key



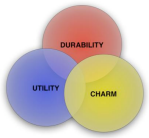
More on the Model

- 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



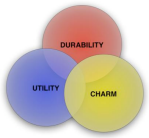
Binding Model and Implementation

- 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



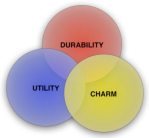
Model Driven Design

- 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



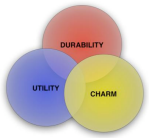
Hands on Modelers

- 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 pseudocode 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



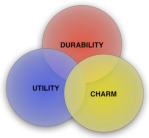
Model Theory

- According to Stachowiak (from Kuhne), a model needs to possess three features:
  - Mapping feature - model is based on an original
  - Reduction feature - model only reflects a (relevant) selection of the original properties
    - Activity to realize this is abstraction
  - Pragmatic feature - a model needs to be usable in place of the original with respect to some purpose



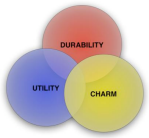
Pragmatic Feature

- A model is information:
  - On something (content, meaning)
  - Created by someone (sender)
  - For somebody (receiver)
  - For some purpose (usage context)



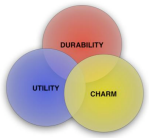
More Models

- A copy is not a model
- And notwithstanding the previous slides a model can also be a theoretical projection of a possible or imaginary system
- Not every transformation is a model!
  - Back to the three features
  - For example test scripts are usually not models



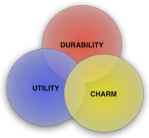
Types of Models

- Token Models
  - Elements capture singular elements of the originals - addresses of a map
  - Can be derived from a higher level model "the map"
  - Useful for capturing system configurations and aspects for simulations
  - Blueprints are token models
  - Not used extensively in model driven development



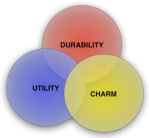
Types of Models -2

- Type Models
  - Attempt to capture universal aspects of an original's elements
  - Instead of particular elements: 15 Frank Sinatra Drive in Hoboken, we state "Lot number, Street and City"
- Token models - singular aspects
- Type models - universal aspects
- Relativism - whether it is a type or token model depends on the original!



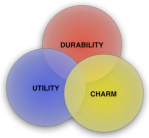
Refactoring Toward Deeper Insight - III

- Correspondence between model and implementation:
  - Challenge is to find a model that reflects a deep (and relevant) understanding of the domain
  - Construct the software so that it is in tune with how the domain expert thinks:
    - Sophisticated domain models are possible and worth it
    - Domain models are developed through an iterative process of refactoring with close interaction of experts and developers
    - Requires sophisticated design skills to implement and use effectively



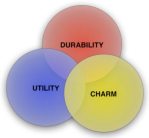
Levels of Refactoring

- Redesigning without changing functionality
- Micro refactoring focuses on code (topic of last few lectures) more mechanical
- Refactoring to design patterns - recognizing, then applying
- Refactoring current design to make it easier to understand
  - Makes design more flexible, easier to understand over time, rather than getting it right at the outset (which never happens)
  - Refactoring as the team achieves new insights into the domain -- refactoring to a deeper model
  - Not only what the code does but also why it does it



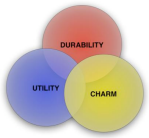
Deep Models

- Noun/verb initial models are naïve, superficial based on shallow knowledge
- "A deep model provides a lucid expression of the primary concerns of the domain experts and the most relevant knowledge, while it discards the superficial aspects of the domain."
- Requires change to accomplish, certain aspects of the model make it easier to change and use
  - Lots of trial and error
- Model Driven Design stands of 2 legs: a Deep Model and a Supple Design, modeling comes to a halt when code is difficult to refactor.



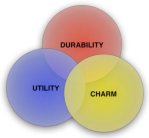
Breakthrough

- Usually an AHA phenomenon -- remember Gestalt Psychology in SSW540?
- Transition to a deeper model usually requires a large shift in conception and a major change to design
- Road to breakthroughs:
  - Be patient, focus on basics, modest improvements lead the way
  - Slowly deepening the model also reveals other design flaws and often this foreshadows a breakthrough
- Payoff -- as model deepens rather than bogging down as complexity is added to complexity in successive releases, the model becomes progressively more straightforward.



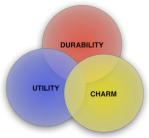
Making the Implicit, Explicit

- Transformations of domain models often occur when concepts hinted at or presented implicitly in design are represented explicitly by one or more discovered relationships
  - It is often implicit due to lack of understanding
  - Often implicit -> explicit fuels a major breakthrough
  - Process usually starts by reorganizing implicit concepts in some form, however crude



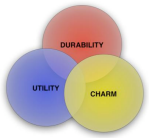
Digging Out Concepts

- Listening to language of the team
- Recognizing awkwardness in design
- Finding contradictions in language of experts
- Mining literature of the domain
- Doing lots of experimentation

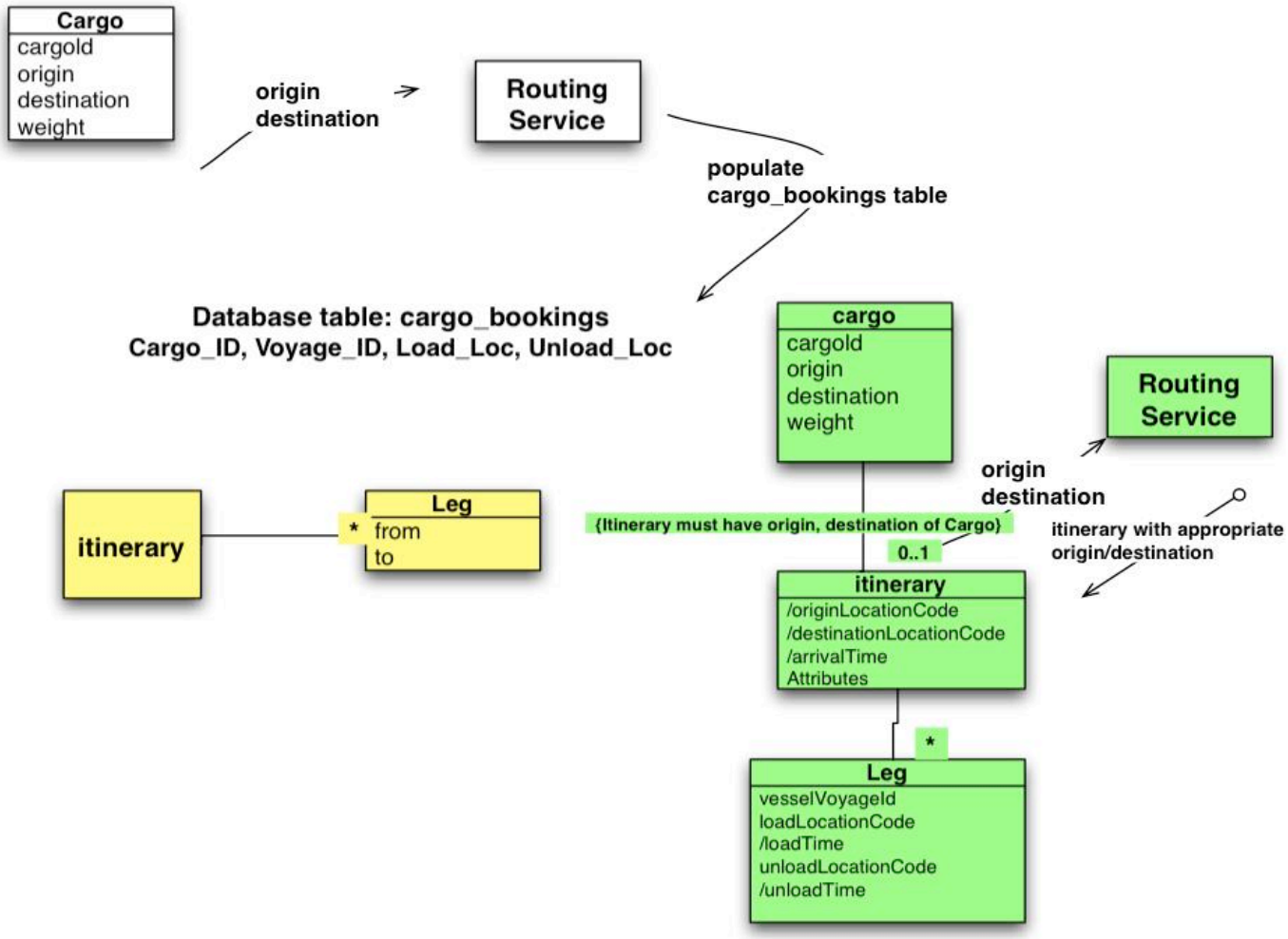


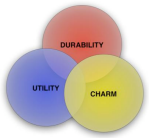
Listening to Language

- Are there terms that succinctly state something complicated?
- Are the experts constantly correcting your word choices?
- Do puzzled looks disappear when you use a certain phrase, do you really understand that phrase?
  - Models your cognitive understanding of the issue
- Hearing a new word produces a lead -> knowledge crunching (intense interaction) -> defining a concept



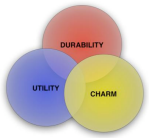
Cargo Example





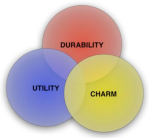
Advantages in Itinerary Role

- Defines interface of routing service, more expressively decouples routing service from tables
- Reduces duplication since itinerary derives loading/unloading times
- Expands Ubi language
- …



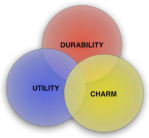
Scrutinize Awkwardness

- Another place to explore is where routines are doing awkward things that are difficult to explain
- You sort of understand them
- Accounting example, adding accruals, decoupling it from payment - provides way to add new, necessary variations of fees and interest



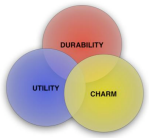
Contemplate Contradictions

- Different domain experts see things different ways
  - Some contradictions are simply there, a function of the domain, but thinking it through can yield interesting insights
- Dig into book knowledge/courses/ training
- Experts will appreciate your attempt to understand their world



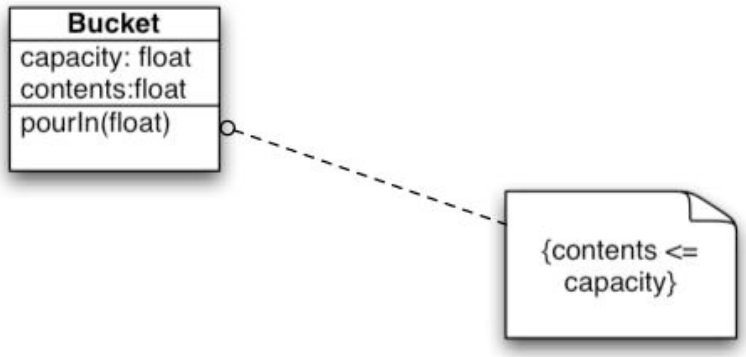
Try, Try Again

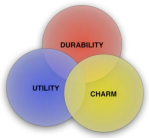
- Lots of trial and error - subteams to explore may help
  - Each change results in deeper insight - need superb teams
- Experimentation/ Design prototypes are essential



Model Less Obvious Concepts

- Explicit constraints often reveal themselves implicitly
- Bucket overflow example!
- Second example makes overflow more explicit

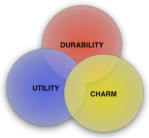




Bucket Example

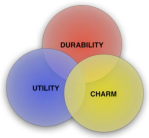
```
class Bucket  {
  private float capacity;
  private float contents;

  public void pourIn(float addedVolume) {
    if( contents + addedVolume > capacity) {
      contents = capacity;
    } else {
      contents = contents + addedVolume;
    }
  }
}
```

```
class Bucket  {
  private float capacity;
  private float contents;

  public void pourIn(float addedVolume) {
    float volumePresent = contents + addedVolume;
    content = constrainedToCapacity(volumePresent);
  }

  private float constrainedToCapacity(float volumePlacedIn) {
    if (volumePlacedIn  >  capacity) return capacity;
    return volumePlacedIn;
  }

}
```



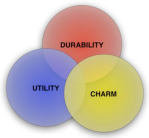
Constraint Warning Signs

- Sometimes a constraint dos not fit into an object and may distort it:
  - Evaluating a constraint requires data that does not otherwise fit into an object
  - Related rules appear in multiple objects forcing duplication or inheritance between objects not strongly related (not otherwise a family)
  - Design and requirements conversations focus on constraints but in the code these constraints are hidden in procedural code
- To repair, factor the constraint to an explicit object or set of objects with relationships

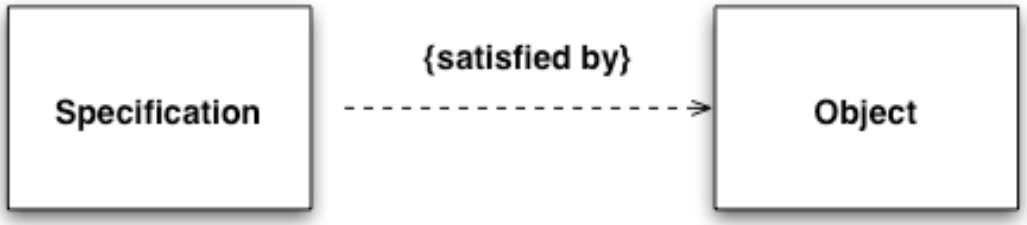


Processes as Domain Objects

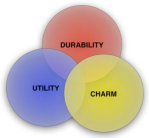
- If the experts discuss a process it should be made explicit, an object providing a service, as opposed to processes that are development mechanisms and encapsulated in a method
- Specification provides a concise way of expressing certain rules, untangling them from conditional logic and making them explicit



Specification

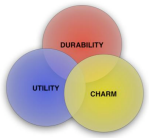


- Mimic logic programming and create specialized rules objects that evaluate to a boolean
- Specification states a constraint on the state of another object
  - Create predicate-like value objects that determines whether an object satisfies some criteria



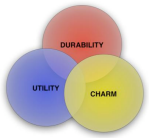
Use Specification

- To validate an object to see if it fulfills some need or is ready for some purpose (preconditions)
- To select an object from a collection (discovering overdue invoices)
- To specify the creation of an object to fit some need

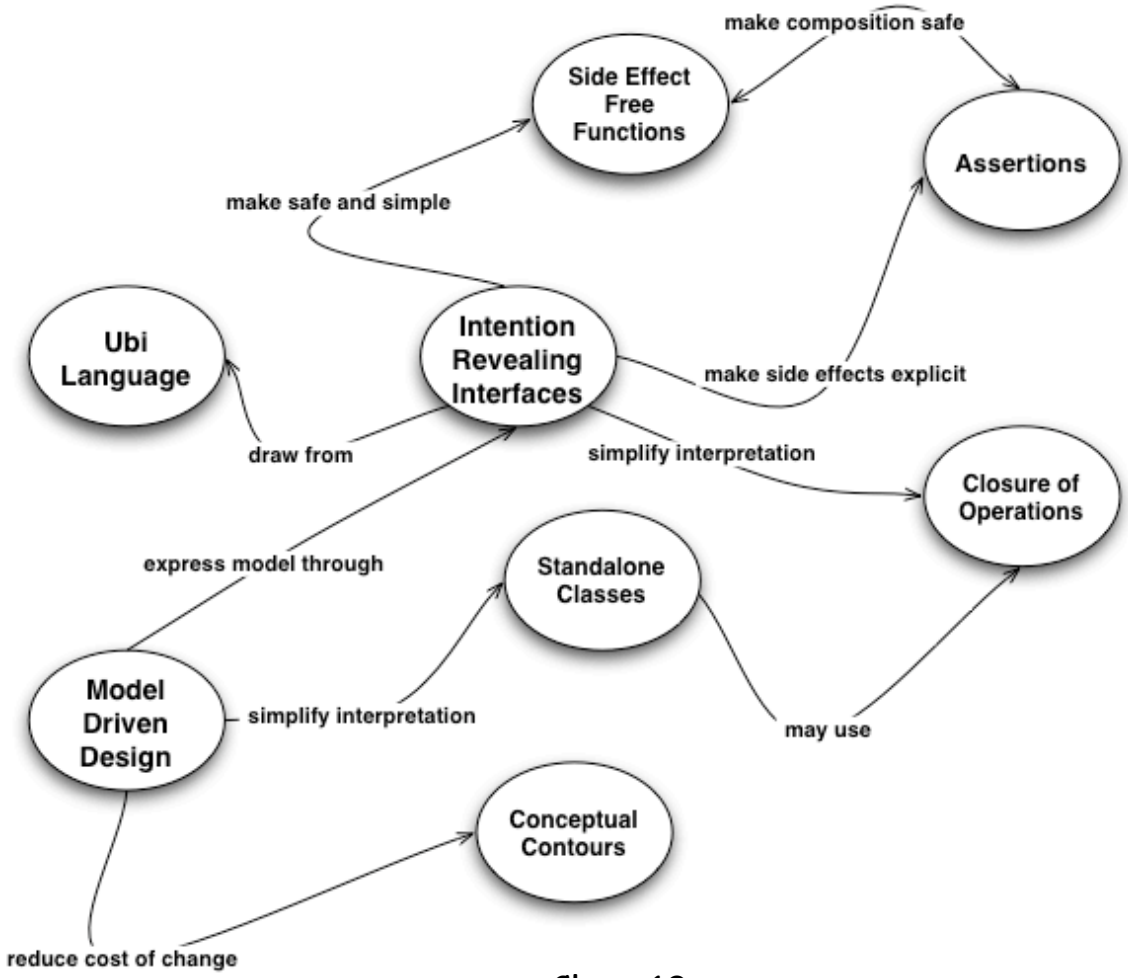


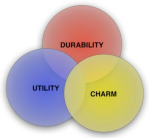
Supple Design

- Software with complex behavior and poor design is difficult to refactor and combine elements
  - Duplication appears to insure that things are done right, beginning of entropy
  - Avoid making changes to the mess and just working around it so that you think you are not creating new problems
- A supple design is a pleasure to work with and maintain - it is the complement to a deep model
- Cultivating a model that captures the main concerns of the domain and shaping a design that permits the developers to put that model to work
- Best designs are simple and simple ain't simple!
- On Simplicity…



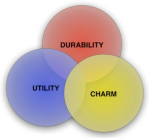
Patterns and Supple Design





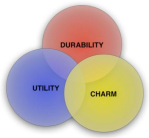
Supple Design - 2

- Must serve the developers that build and change the design
- In reality only parts will be supple and hang in there, first attempts are not usually supple
- No prescription for supple design but the patterns will help



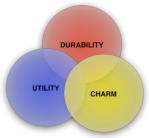
Intention Revealing Interfaces

- Previously we discussed implementing rules and calculations explicitly
- If the interface does not tell the developer what is necessary, the developer digs - not good
  - Same with a reader of the code
  - Most of the value in encapsulation is lost
  - You are leaving the understanding of the true purpose of the object/aggregation to chance
  - Corrupts conceptual design and encourages the growth of entropy



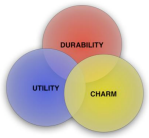
IRI-2

- Elements must be named so that the names reflect the concepts for the classes and methods - use the Ubi language
- Relieves the developer of understanding the internals
- Write a test for the behavior before creating it to force one into implementer mode (sound familiar?)



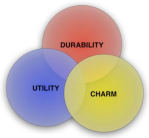
IRI - 3

- Tricky/complex mechanisms should be encapsulated behind abstract interfaces that speak of intentions (what) rather than means (how)
  - State relationships and rules - not how they are enforced
  - Describe events and actions - not how they are carried out
  - Formulate equations - not the numerical method to solve it
  - Pose the question to answer - not how it will be answered



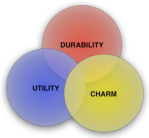
IRI - 4

- Counter point - Open Implementation, Gregor Kiczales
  - Use module's primary interface when sufficient
  - BUT if not acceptable, control the modules implementation through a meta interface
  - Key is to separate functionality (black box) and implementation strategies and functions



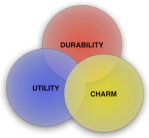
Side Effect Free Functions

- Any change in the state of the system that affects future operations
  - Computer science definition is a bit stricter - and effect on the state of the system
- Operations that return results without side effects are called functions
  - Functional programming, caml, is about a side effect free language
- Functions are much easier to test than operations that have side effects (why?)
  - Functions lower risk



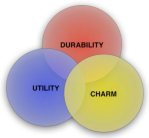
Commands

- On the other hand commands (aka modifiers) are operations that affect some change to the system--produce side effects.
  - Queries obtain information by accessing data in a variable possibly doing a calculation based on it
- Commands can not be avoided but can be tamed in 2 ways:
  - Keep commands and queries segregated, methods that cause changes should not return domain data: get_account vs deposit_into_account. Do all queries and calculations in methods that cause no observable side effects
  - May be alternative models that do not call for existing object to be modified - instead create a new Value Object in answer to a query, hand off and forget
    - Value Objects are immutable after initialization, all their operations are functions



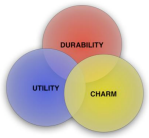
Assertions

- Makes side effects explicit and easier to deal with
- Commands often invoke other commands and you must understand the whole chain in order to understand what is happening
  - In this case the values of encapsulation and abstraction are lost, information is not hidden
- One way to solve this is by contracts



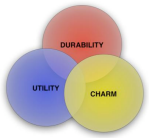
Contracts

- In contracts preconditions and post-conditions (side effects are defined)
  - If you trust the post condition description you do not have to understand how a method works
- State post conditions and invariants
  - Write automated tests for them
  - Write into documentation
  - Code should make it easier for developer to infer assertions



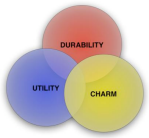
Conceptual Contours

- When elements of a model or design are embedded in a monolithic construction- functionality gets duplicated (old style fortran, basic sans subroutines)
- Other extreme is breaking down classes and methods too finely result in in lots of moving parts - novice OO
- Goal is simple set of interfaces that combine logically to make sensible statements in Ubi language
- Conceptual contours are divisions of the domain -- the design is aligned with the underlying concepts of the domain, therefore if new domain knowledge if new knowledge is acquired/needed if fits into current design



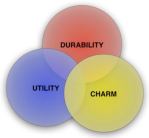
In Summary

- Intention revealing interfaces permit presenting objects as units of meaning rather than mechanisms in the program
- Side effect free functions and assertions make it safe to use these units
- Emergence of conceptual contours stabilizes parts of the model and makes units more intuitive to use, combine and enhance since it relates to the actual domain



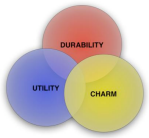
Standalone Classes

- Modules and aggregates limit the web of interdependencies
- In an important subset of concepts the number of dependencies can be reduced to 0. So the class can be fully understood by itself (integers)
- Low coupling is fundamental to model design when you can eliminate all other concepts, coupling = 0
  - Candidates usually Value Objects



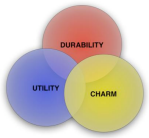
The Rest

- Closure of operations - return type matches calling type
- Declarative design - write programs as specifications (rule based a candidate) usually not expressive enough
- Domain Specific Languages - tiny languages, requires high skill, compatibility issues when language needs to be changed
- Declarative style of design - use when you can



How - Angles of Attack

- Carve off sub domain
- Draw on established formalisms for the domain
- Some keys:
  - Live in domain
  - Keep looking at things in different ways
  - Maintain a dialog with experts
- Initiation (something is missing, not right) then onto exploration in sub domain.



Other References

- Portland Design Repository
- Gamma, Helm, Johnson & Vlissides Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- Maeda, J. The Laws of Simplicity, MIT Press, 2006.
- Kuhne, T. "What is a model" Dagstuhl Seminar Proceedings.
- Conway, Damian, http://www.perl.com/pub/a/2005/07/14/bestpractices.html
- Budgen, D. Software Design, Addison-Wesley, 2003, ISBN: 0-201-72219-4