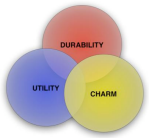
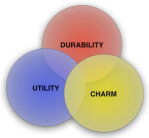


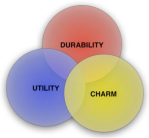
Class 7 SSW565

Gregg Vesonder
Stevens Institute of Technology
©2009 Gregg Vesonder



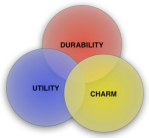
Roadmap

- Logbook
  - Me
- Agile Methodology
- Review Basic Software Design
- Part 1 in Evans
- Readings next class, chapters 5 thru 7 in Evans, Domain Driven Design



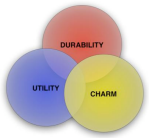
Key Dates

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



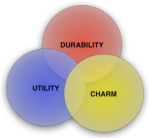
Clarification

- The Mid Term

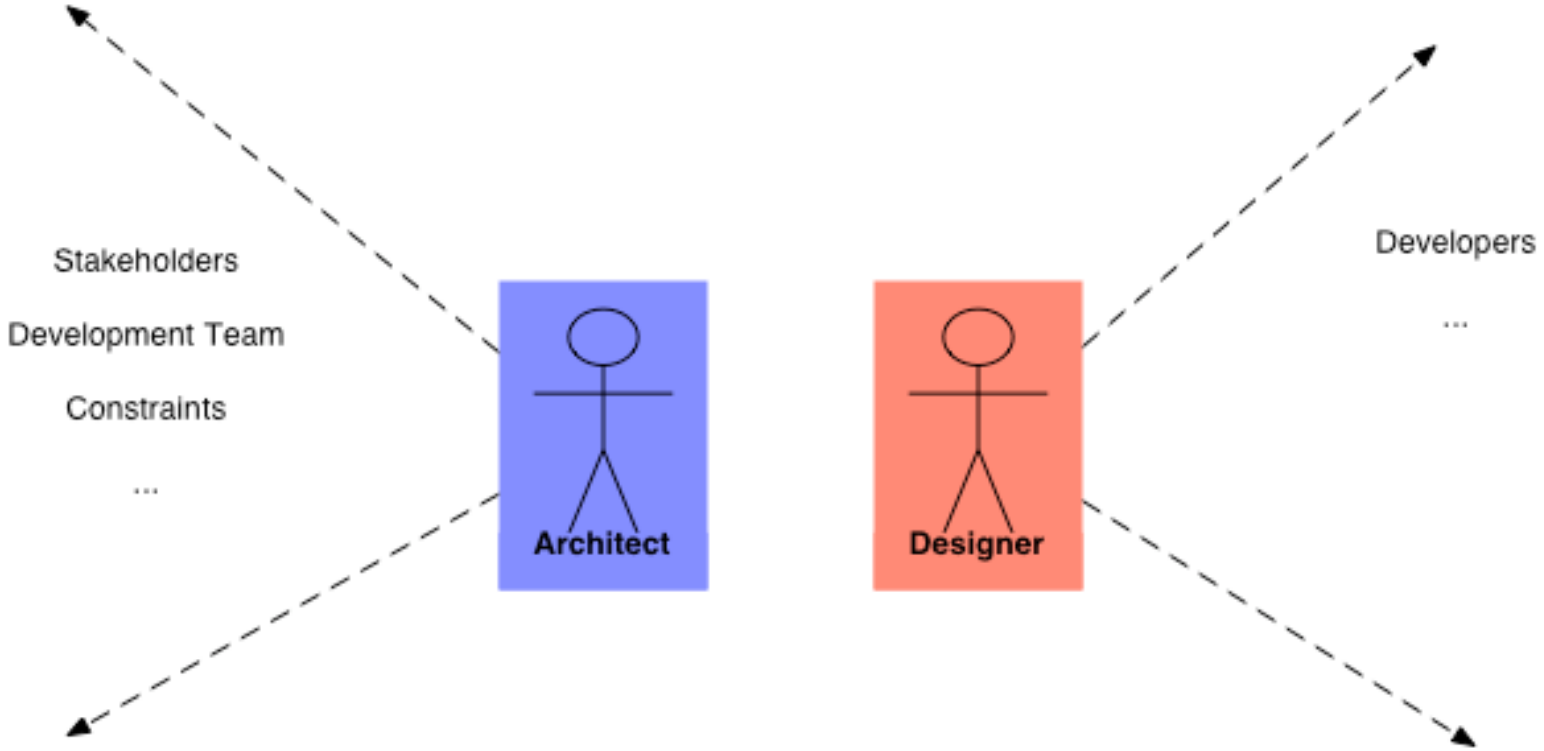


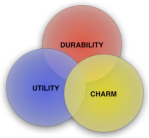
Logbook

- Flocking Behavior and Design
- Your turn



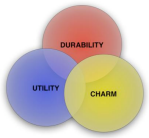
Architecture and Design





The Context and Review

- Finished architecture and the architect is around and may contribute to design, certainly the architect oversees design
- Moving into the world of Agile processes, why?
  - Forefront of software technology
  - Heavily draws on modern design principles: design, teaming with customer, prototyping, knowledge engineering, code is number 1, light documentation
  - May not even have an architect
- Ties together much of what we discussed in 540 and so far in 565
- Take aspects and apply to your projects
- DO NOT FEAR THE CODE - the team has to read it



http://agilemanifesto.org

Agile Manifesto

Manifesto for Agile Software Development

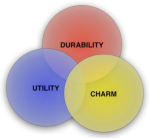
We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

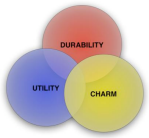
Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.



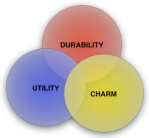
Focus on Light Methodology

- Customer value
- Creating a culture of innovation, creation and rapid delivery
- Appealing to skilled, talented staff
- Facilitating collaboration, knowledge sharing and decision making
- Reducing, time, cost and defect levels significantly.



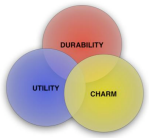
Some Light Methodologies

- Extreme Programming - Kent Beck
- Crystal Methods - Alistair Cockburn
- Lean Development - Bob Charette
- SCRUM - K. Schwaber, J. Sutherland
- Adaptive Software Development - Jim Highsmith
- …



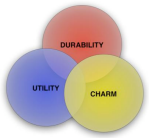
XP

- XP's basic premise - coding is THE KEY activity
- Geared for small to medium sized teams
- Calls for implementing highest priority features first
- Customer is integral part of the team
- Define smallest code release possible
- Programmers accept responsibility for estimating and completing work - feedback
- Encourages human contact, incorporates method for staff turnover



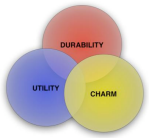
Underpinnings of XP

- If code review is good, do it all the time, pair programming
- If testing is good, everyone will test (unit testing), customers (functional testing)
- If design is good, it is every day for everyone (refactoring)
- If simplicity is good, we'll leave system with simplest design that supports the functionality - the simplest thing that can possibly work
- If architecture is important everyone does/defines/refines architecture all the time
- If integration testing is important then we will integrate and test several times a day
- If iterations are important we'll make iterations really short, minutes and hours, not weeks and months!



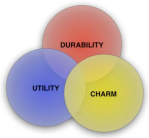
Differs from other methods

- Short cycles provide early, concrete and continuing feedback
- Incremental planning that quickly generates an overall plan that evolves
- Responds to changing needs by flexibly scheduling implementation of functionality
- Reliance on automated tests to catch defects early, monitor progress and allow system to evolve
- Reliance on oral communication, tests and source code to communicate system structure and intent
- Reliance on evolutionary design process throughout development
- Reliance on close collaboration of programmers with ordinary skills
- Reliance on practices that mesh with short term instincts of programmers and long term interests of project



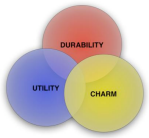
A Day in the Life of an XPer

- Stand up meeting begins day
- Stack of task cards provides tasks
- Invite programmer to be your partner
- Develop together - both at the screen, mouse and keyboard concerned with implementation, other more globally
- First build tests, run tests
- Develop program, assess design, collaborate
- Test
- Programming pairs evolve design of the system - they can change everything!



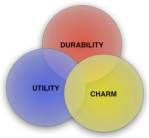
Four Values of XP

- Communication: unit testing, pair programming, and task estimation cause programmers, customers and managers to communicate
- Simplicity - better to do a simple thing today and change later, than a more complicated thing that will not be used
- Feedback - unit tests, customers write stories (feature descriptions), programmers estimate -- when all the tests are run you are done
- Courage - within the context of the first three values - "go like hell!" However, courage by itself is "just plain, bad hacking!"
- Other comments:
  - XP resembles hill climbing local optimas require large change
  - Need a real team that respect each other and have passion for what they do



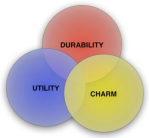
Fundamental Principles of XP

- Rapid feedback
- Assume simplicity
- Incremental change - smallest change that makes a difference
- Embracing change
- Quality work - excellent or insanely excellent



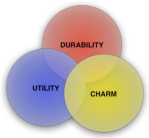
Less Central Principles

- Teach learning
- Small initial investment
- Play to win, rather than playing not to lose
- Concrete experiments - especially regarding requirements
- Open honest communication
- Work with people's instincts, not against them - "XP matches observations of programmers in the wild"
- Accepted responsibility
- Local adaptation
- Travel light
- Honest measurement



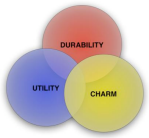
Key Aspects of XP

- **Whole team** - development + customer
- **Metaphor** - everyone use common analogy in discussing the system, e.g., desktop metaphor -Scandinavian School
- **The Planning Game** - specify the next step of development and, as project progresses, provides better and better picture of what will be delivered. **User stories** lead to development cost estimates, leads to client assigning priorities, leads to evaluating estimates for the next round
- **Simple design** - design should only incorporate at best next iteration - if it becomes complex, refactor
- **Small releases** - every development cycle (~ 2 weeks) client gets new software.



Key Aspects of XP - 2

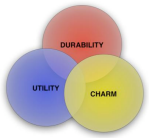
- **Customer tests** - customer develops acceptance tests based on user stories, automated and used frequently by development team
- **Pair programming**
- **Test-driven development** - test first, add to suite
- **Design improvement** - refactoring and small improvements in design, simple design
- **Collective code ownership** - source code control, "refrigerator in frat house" phenomenon (anything you put in, you should not expect to see next time)
- **Continuous integration**
- **Sustainable pace**
- **Coding standards**



Differences

- TSP has detailed process control, quality and performance metrics
- TSP has large number of scripts, forms roles and exit criteria
- TSP has established reports for tasks and phases and keeps history of activity
- TSP formal/contractual relationship with customer

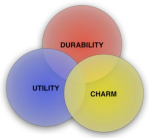
- XP metrics are product oriented estimating progress and future iterations -- performance and quality responsibility of pairs
- Few guidelines and strict practices
- XP reporting informal and has historian but rather than activity focus it is a post mortem focus (newspaper vs analysis)
- XP collocated, collegial relationship with customer



The Schedule

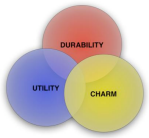
- Part 1: Putting the Domain Model to Work - today
- Part 2: The Building Blocks of Model-Driven Design – 6/25
- Part3: Refactoring Toward Deeper Insight – 6/29
- Part 4: Strategic Design and Refactoring begins– 7/6
- Refactoring Book – 7/13
- Catch-up, Special topics and review - 7/20

First a bit about "traditional" design



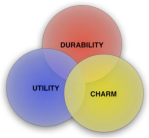
SOFTWARE DESIGN: A wicked problem

- No definite formulation of a wicked problem - design overlaps other stages
- No stopping rule - dangerous cycle
- Not true or false - no 8 ball, but satisficing
- Every wicked problem is a symptom of another problem - resolving one may result others
- May also term this ill-formed problems - Newell and Simon
- Suggests that we pay equal attention to the human system - the Scandinavian school



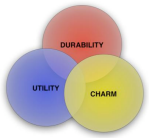
On Design

- Design Problem- decompose a system into parts, each part having a lower complexity than the system as a whole and the interaction between the parts is not complicated. These parts and their interaction solve the user's problem .
- Architecture is the characterization of the design process.
- Five elements affect the quality of the design: abstraction, modularity, information hiding, complexity and system structure



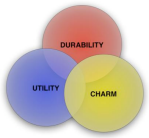
Design - 2

- A module is an identifiable unit in design
- Design features we are most interested in are those that facilitate maintenance and reuse:
  - Simplicity
  - Clear separation of concepts into different modules
  - Restricted visibility, locality of information, "secrets of the module"



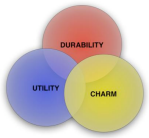
Abstraction

- Concentrate on essential issues and ignore (abstract from) details irrelevant at this stage
- A problem is decomposed to sub-problems with major tasks recognized by their description and the verb (read, sort, process)
- Essence of main program with subroutine architecture style
- Procedural abstraction is the name of the procedure designating the actions
- Data abstraction- finding a hierarchy in the program's data and data typing set of objects and operations on them



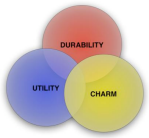
Modularity

- 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



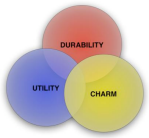
Yourdon and Constantine, 7 (+1) levels of Cohesion

- Levels are of increasing strength:
  - Coincidental- modules grouped in haphazard way, no relation
  - Logical - logically related tasks that do not call each other or pass data between each other - all output routines
  - Temporal - various independent components activated at same time - initialization
  - Procedural- group of components executed in a set order
  - Communicational - components operate on the same temporal data
  - Sequential- output of one serves as input to another
  - Functional all components contribute to a single function in the module
  - + data - modules that encapsulate an abstract data type



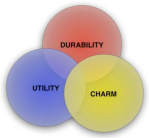
Coupling

- Tightest to loosest (worst to best):
  - Content- one module directly affects working of another
  - Common- 2 modules have shared data
  - External modules communicate through external media, a file
  - Control - one module directs execution of another by passing necessary control information via flags
  - Stamp - complete data structures are passed from one to another
  - Data - only single data passed between modules



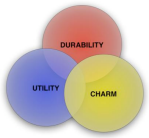
Coupling and Cohesion

- 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.



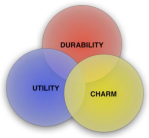
Information Hiding

- 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



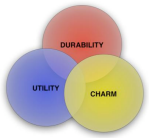
Complexity

- Attributes of the software that affect effort needed to construct or change a piece of software
- 2 classes of complexity metrics
  - Size based - KLOC
  - Structure based - complicated control or data structures

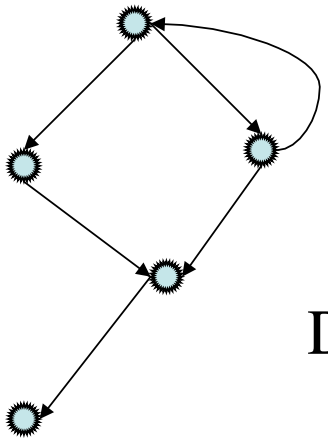


System Structure

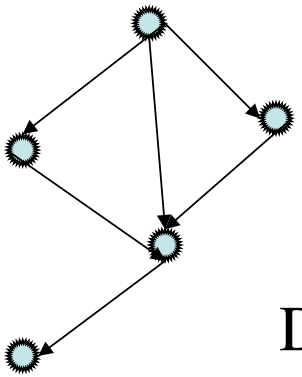
- Types of intermodule relations such as A contains B, A follows B, A delivers data to B, A uses B
- The amount of knowledge each uses about the other should be kept to a minimum
  - Information flow should be limited to procedure calls - no Common data structures
- Graph depicting procedure calls is a call graph - we can measure attributes related to the shape of the call graph



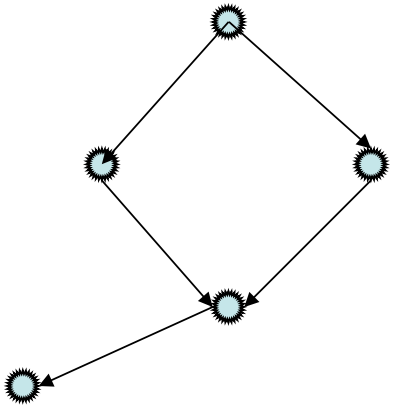
Module Hierarchies



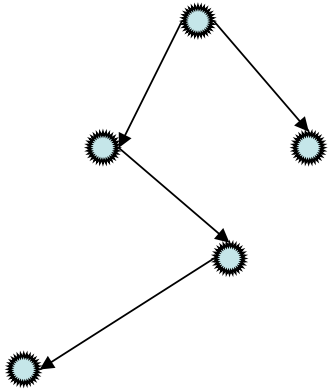
Directed graph



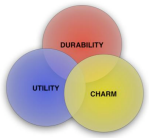
Directed acyclic graph



Layered graph

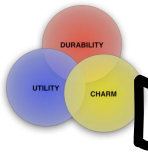


tree



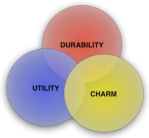
Graph analysis

- Some measures:
  - Size - number of nodes and edges
  - Depth - longest path from root to leaf
  - Width - maximum nodes at some level
- A good design should have a tree like call graph
  - One measure of complexity is to assess tree impurity
  - Remove edges until you get a tree
  - Tree impurity = # of extra edges / maximum # of extra edges
  - If 0 graph is a tree, if 1 it is a complete graph
  - But trees are not always desirable, does not permit reuse
- Fan in /fan out measures indicates spots deserving attention, e.g., if a module has high fan in it may indicate little cohesion, excessive increase in information flow from one level to next may indicate missing level of abstraction



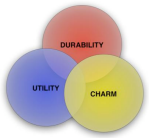
Design Heuristics for Modularity (Pressman) - start here

- Evaluate the first iteration of a program structure to reduce coupling and improve cohesion
- Attempt to minimize structures with high fan out; strive for high fan in as depth increases
- Keep the scope of effect of a module within the scope of control of that module
- Evaluate module interfaces to reduce complexity and redundancy and improve consistency
- Define modules whose function is predictable, but avoid modules that are overly restrictive - balance!
- Strive for controlled entry modules by avoiding pathological conditions (branches or references into the middle of a module)



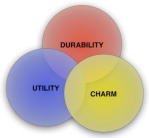
Design Methods

- Functional decomposition - next slide
- Data flow design - functional decomposition with respect to flow of data. Component is a black box transforming some input stream to an output stream
- Design based on data structures - given a correct model of data structures, design of the program is straightforward
- Object-oriented design - later



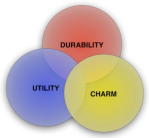
Functional Decomposition

- Intended function decomposed into sub functions and continues downward
- Start from user end it is top-down, primitives, bottom-up
- Parnas method:
  - Identify sub systems, start with a minimal subset and define minimal extensions (incremental development)
  - Apply information hiding principles
  - Define extensions step by step
  - Apply "uses" relation and try to develop a hierarchy
  - Layered approach, use only components at the same or lower level



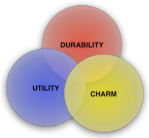
Design Documentation

- IEEE 1016
- Seven user roles for the design documentation:
  - Project manager
  - Configuration manager
  - Designer
  - Programmer
  - Unit tester
  - Integration tester
  - Maintenance programmer



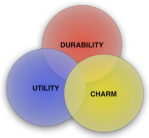
View on Design

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



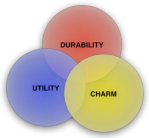
Verification and Validation

- Inspection and walk throughs, reading and critiquing text
- Formal techniques for problem areas
- Prototypes
- Test cases on each of the modules



The Context

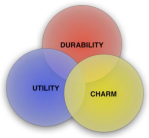
- Finished architecture and the architect is around and may contribute to design, certainly the architect oversees design
- Moving into the world of Agile processes, why?
  - Forefront of software technology
  - Heavily draws on modern design principles: design, teaming with customer, prototyping, knowledge engineering, code is number 1, light documentation
  - May not even have an architect
- Ties together much of what we discussed in 540 and so far in 565
- Take aspects and apply to your projects
- DO NOT FEAR THE CODE - the team has to read it



Back to design!

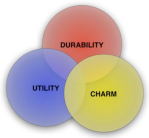
What's a Model

- 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



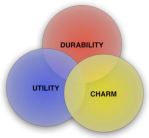
Model's Role in Design

- The model, the design and the implementation shape each other -- model and implementation are closely bound (essential)
- Model is the backbone of the language used by ALL team members (experts and developers) -- Ubiquitous Language
  - It is the bridge between domain experts and developers
- Model is distilled knowledge
  - Structure domain knowledge
  - Highlights elements of interest



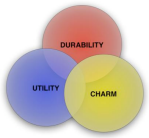
Heart of Software

- Solve domain related problems for its users
  - Discover the conceptual integrity, manage complexity, recall last chapters of Brooks
- Developers have to steep in the domain
  - Messiness of most domains is the technical challenge … inhomogeneity of technical understanding
- Create a lucid model that cuts through complexity, is a medium for communication and sustains development



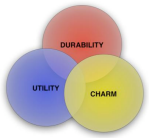
Ingredients of Effective Modeling

- Entry conditions to modeling:
  - Architecture with constraints
  - High level design or initial prototype
  - XP list of task cards (maybe)
- Binding the model and the implementation
  - Use the prototype
- Cultivate a language based on the model - Ubiquitous Language (Ubi Language)
  - Bridge between expert and developer
  - Common, shared vocabulary
- Model is knowledge rich with behavior and inferred rules (business rules) NOT a data description/schema



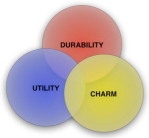
Effective Modeling -2

- Model is then distilled/tuned, important concepts added, extraneous ones dropped
- Moves to Brainstorming and Experimenting with entire team
  - Prototyping, a feedback loop through implementation
- Sift through explosion of data and knowledge
  - Where established architecture would help
  - Try one organizing idea after another
  - Trial and error
  - Assess coverage of the model to the information



The Team

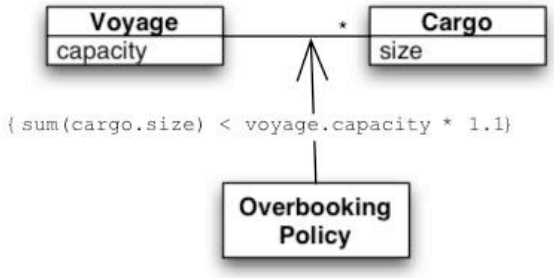
- Developers + Domain Experts(SMEs)
- Why not other software process models
  - Waterfall model lacked feedback
  - Iterative model
    - Feedback, but no abstraction
    - Only discovering what program should do, not what lies underneath - essential to know to build the right system and plan for future
- Development learns important principles of business rather than representing requirements mechanically
- Virtuous cycle of evolution - continuous learning
  - Knowledge in the model goes beyond find the nouns and verbs in requirements
  - Discover inconsistencies in business rules



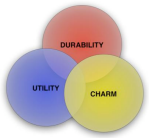
Overbooking Policy Example


```
public int makeBooking(Cargo cargo, Voyage voyage) {
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
```


```
public int makeBooking(Cargo cargo, Voyage voyage) {
    double maxBooking = voyage.capacity() * 1.1;
    if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
      return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
```

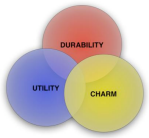



```
public int makeBooking(Cargo cargo, Voyage voyage) {
    if(!overbookingPolicy.isAllowed(cargo, voyage))
      return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
```



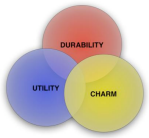
Takeaways From Overbooking

- Stating the overbooking rule as a policy makes it explicit
- Example of the Strategy pattern
- Demonstrates that a domain model & design can be used to secure and share knowledge. Advantages:
  - All share knowledge
  - Knowledge represented in model and code is clear to everyone -- it is traceable



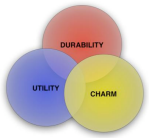
Design Patterns

- Portland Design Repository - http://c2.com/ppr/
- Motivation came from a "real" architect, Christopher Alexander-
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution over a million times without ever doing it the same way twice."
  - Referring to buildings and towns, e.g. couple's realm, children's realm, sleeping to the east, …



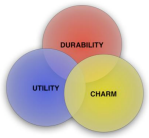
The Pattern Structure

- Pattern name
  - Pattern name and classification
  - AKA
- The problem
  - Intent
  - Motivation
  - Applicability
- The solution
  - Structure
  - Participants
  - collaboration
  - Implementation - can be ignored, pseudo code at best
  - Known uses
  - Related patterns
- The consequences



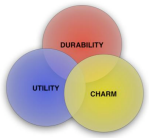
Terminology

- Responsibilities:
  - The knowledge an object maintains (variables, algorithms, ….)
  - The actions an object can perform
  - Sense of purpose to the object and place in the system
  - Represent only publicly available services
  - In requirements spec:
    - Often verbs
    - Everywhere information is mentioned, since information that some object must maintain and manipulate
- Object as server when it fulfills the request of another object
- Contract - is a list of services an instance of one class can request from an instance of another
- Classes fulfill at least one responsibility, name suggests the responsibility



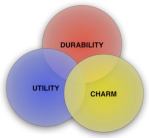
Categories of Patterns

- The Gang of Four, Gamma, Helm, Johnson & Vlissides
- Creational - abstract instantiation, strive for independence
- Structural Patterns - how classes and objects are composed to form larger structures--uses inheritance
- Behavioral - algorithms and assignment of responsibility, describe objects and communication



The Patterns

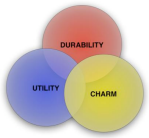
- Creational
  - Abstract factory
  - Builder
  - Factory method
  - Prototype
  - Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorater
  - Façade
  - Flyweight
  - Proxy
- Behavioral
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor



Now back to describing The pattern!

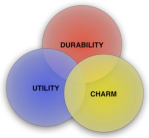
Strategy

- Pattern Name - strategy, aka policy
- Problem - different variants of an algorithm are needed, need to hide data an algorithm uses from clients.
- Solution - define classes for each strategy and an interface that is common to all supported strategies.
- Consequences - provides a way to represent hierarchies of related algorithms, provides a way to vary an algorithm supporting a class dynamically. Clients must be aware of the strategies and could increase number of objects unless you implement as stateless objects using a flyweight approach.



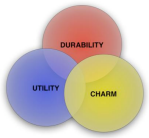
What IS the Model

- Not limited to the UML
  - Written text
  - Informal diagrams (boxes and lines)
  - Casual conversation
- Requires a common language, a Ubiquitous Language
  - Shared team language >>>> glossary
  - The team, especially the developers, must be committed to this language



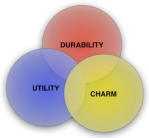
Ubi Language

- Domain model serves as the backbone for the common language
- Vocabulary = names of classes + prominent operations
- Patterns are included
- "semantics" of model = meaning of the words and usage/ phrases/jargon of the team
- Continue to use the model + language to express everything, if you can't then change model and language!



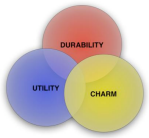
Ubi Language -2

- Use same language in everything: writing, diagrams and speech
- Resolve confusion of terms in conversation
  - Domain experts should object to awkwardness or inadequacy in language
  - Developers object to ambiguity or inconsistency
- Language expresses the large scale structure of the system -- used to discuss the architecture, maintain and nurture the conceptual integrity
- Aid to developers, speaking about objects easier than discussing tables
- Modeling out loud, need to focus on relationships and interactions among objects

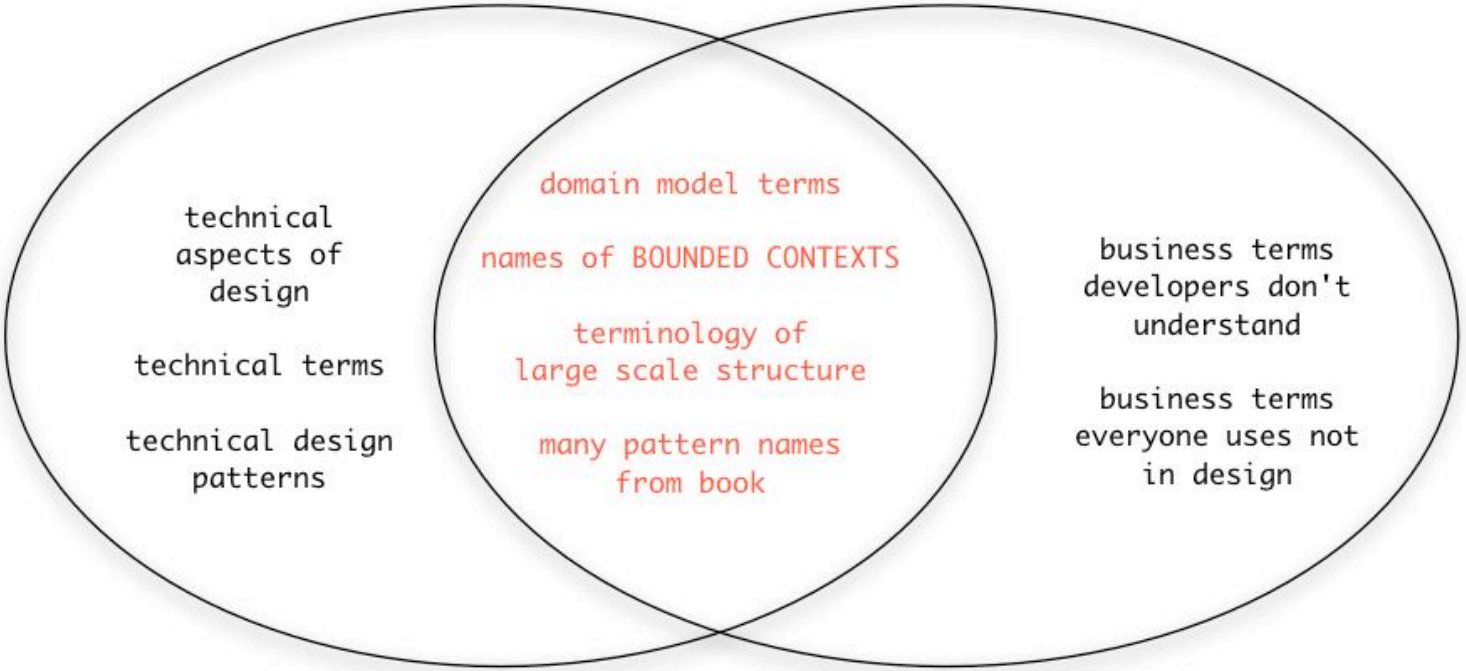


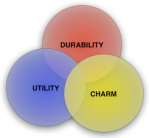
One Team, One Language

- "If sophisticated domain experts do not understand the model, there is something wrong with the model."
- Test model by walking through the scenarios
  - Use language of the model to specify use cases!
  - Use language of the model to specify acceptance tests!
- Ubi language represents the intersection of the jargons



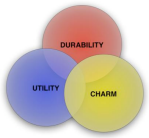
Intersection - 2.3





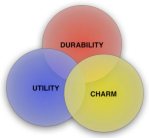
Documents and Diagrams

- Simple, informal UML diagrams can anchor a discussion
  - And they are not difficult to produce
- "The trouble comes when people feel compelled to convey the whole model or design through UML."
- Behavior of the objects and the constraints are not so easily illustrated -- too much of the tree, not enough of the forest
- The model is not solely the diagram
- The vital detail of the design is captured in the code
- Key premise: if you can't code, you can't design



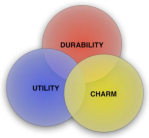
Written Design Documents

- 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



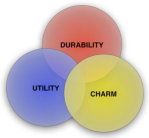
More on the Model

- 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 byt 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



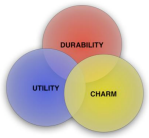
Binding Model and Implementation

- 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



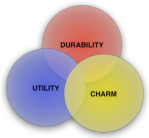
Model Driven Design

- 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



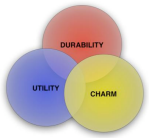
Object Design

- 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

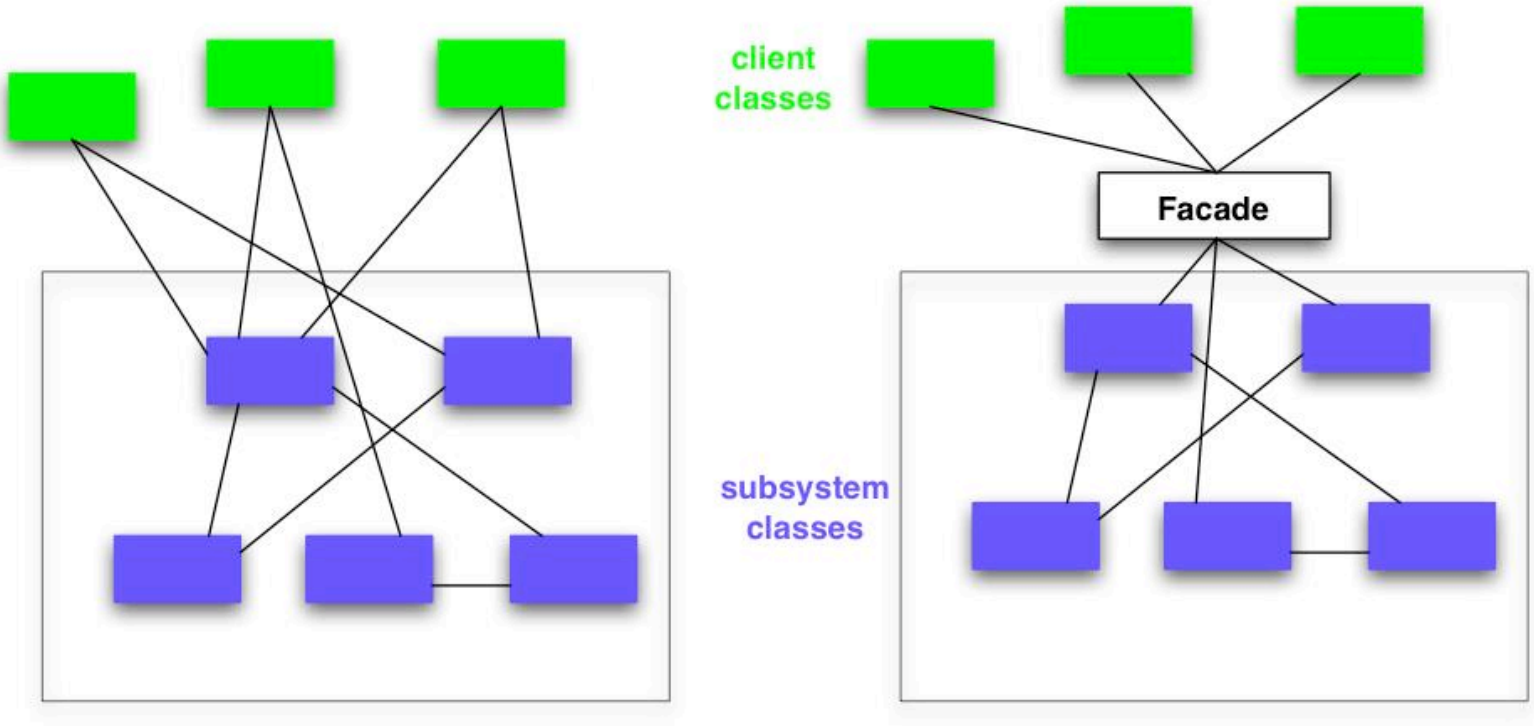


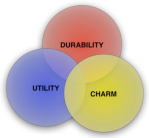
Facade

- **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.


DURABILITY
UTILITY
CHARM

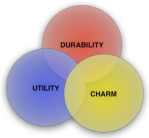
Facade


client
classes
Facade
subsystem
classes



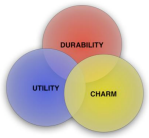
Hands on Modelers

- 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



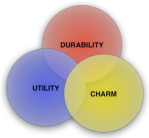
Part 2 - Building Blocks

- 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



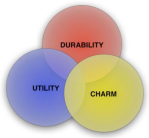
Other References

- Portland Design Repository
- Wirfs-Brock, R. Wilkerson, B. & Wiener, L. Designing Object-Oriented Software, Prentice Hall, 1990.
- PLoP - Pattern Language of Programs Conference
- Gamma, Helm, Johnson & Vlissides Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- R.S. Pressman, Software Engineering a Practitioner's Approach, McGraw-Hill, 5th edition, 2001, ISBN: 0-07-365578-3.



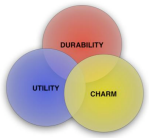
Abstract Factory

- 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.



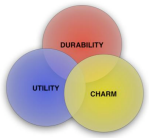
Abstract Factory-2

- Consequences - ability to develop code once 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 factor, 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 bette have those products.



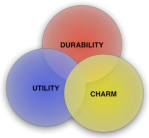
Builder

- 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. Go over maze implementation.
- Consequences - vary a product's internal representation, better control over the construction process and therefore the internal structure of the product.



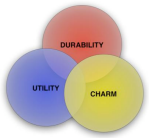
Factory Method

- 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.



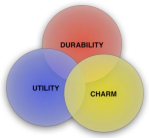
Factory Method -2

- 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.



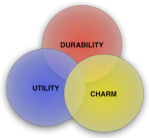
Prototype

- Pattern Name - prototype
- Problem- need to define "classes" at runtime, dynamic creation or classes do not have state. Want to avoid creating class hierarchies.
- Solution - provide the effect of different classes by cloning, making copies of objects with different parameters (state). These parameters could even include bitmaps or fonts or … . Great when "classes" are instantiated at runtime or when you do not want to build class hierarchies or when there is not to many combinations of state. In effect the combination of parameters, when registered as a prototype becomes a new classes that you can freely create more instances of. These prototype instances become "classes"



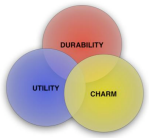
Prototype-2

- Consequences - again hides concrete classes from client. Can add products at runtime by combinations of parameters and these instances become prototypes. For example you can refer to a combination of parts dynamically as a prototype and this becomes a new class that you can instantiate. Must be able to support a deep - true copy that is cloning all instance variables too rather than sharing them.



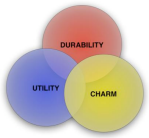
Singleton

- 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++).



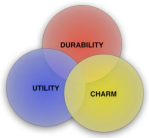
Adapter

- Pattern Name - adapter aka wrapper (but not quite the same)
- Problem - interface to a class is not interface client expects, but you want to use it. You want to create a class that works with classes that have not been designed but may have different interfaces. Translates.
- Solution - two ways, class adapter uses multiple inheritance to adapt one interface to other or create a parent class that defines interface for subclasses, known as object adapter. Basically whether the adapter interface is defined high in the class hierarchy or low in the class hierarchy.



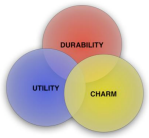
Adapter - 2

- Consequences - for class adapter works only for a class since you commit to an Adaptee class. Lets adapter override adaptees class, introduces only one object. For object adapter a single adapter works with a lot of class that inherit the interface. Overriding is harder.



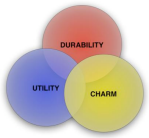
Bridge

- Pattern Name - bridge aka handle, body
- Problem - separate abstraction from implementation by defining what needs to be available (the abstraction) but not the implementation. Therefore we need to bridge the abstraction from the implementation for a particular use.
- Solution - define the abstraction that has a reference to the implementation, a class called implementor that refers to the necessary concrete implementations.



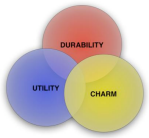
Bridge - 2

- Consequences - implementation is not bound permanently to interface and therefore can be changed, binds one of a few or many implementations, eliminating compile time dependencies. Hides implementation details since the clients refer to abstract interface.



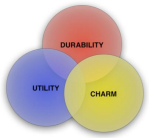
Composite

- Pattern Name - composite
- Problem - need to represent part whole hierarchies and you do not want to deal with all of the objects (parts) in the composition.
- Solution - component handles interactions with composite structure. The component transmits operations to either a composite or and actual part (aka leaf) and if part does it or if composite, composite transmits to its parts.
- Consequences - client code does not need to know if it is a part or a composite, interacts same way, so makes the client simple. This makes it easier to add composites or parts. Sometimes is to general and does not provide the ability to restrict what parts can be used.



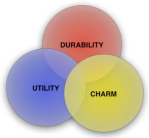
Decorator

- Pattern name: Decorator, a.k.a. wrapper, adorner (MacApp)
- The problem: want to add additional responsibilities to an object not to the entire class. The usual way is to use inheritance, but inheritance is one size fits all every object of the new class will have the responsibilities. Alternative is to enclose component in another object that adds feature.
- The solution - So decorator wraps object in another object and this can be done recursively to add an unlimited number of responsibilities (features). It still is transparent to all the wrapped objects responsibilities.



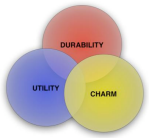
Decorator-2

- The Solution (cont'd)
  - When to use: 1) to add responsibilities to individual objects w/o affecting other objects, 2) withdrawable responsibilities 3)when extension would cause and explosion of subclasses to support every combination -- borders are a good example
  - Changes the objects skin
- The consequences
  - You can "snap-on" responsibilities - even duplicate (e.g.,for double borders)
  - Unlike inheritance, objects do not pay for features they do not use
  - But a "decorated" object and an object it wraps are not identical and you cannot rely on object identity
  - Lots of little objects that look alike.



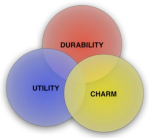
Decorator-3

- Other
  - Provides a way to emulate pipe and filter with each wrapper potentially becoming another filter on a base stream class.
  - Related … especially adapter which does some traditional aspects of wrapper, I.e., changing an objects interface



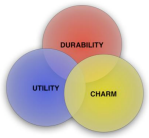
Facade

- 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.



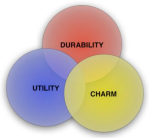
Flyweight

- 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.



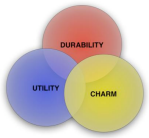
Flyweight - 2

- 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).



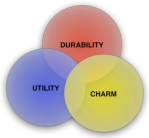
Legacy Wrapper - gtv

- Pattern Name - legacy wrapper, aka wrapper
- Problem - cost of reimplementing legacy systems is prohibitive yet there is a need to make these systems compatible with technology of newer systems.
- Solution - create a wrapper that invokes all the methods of the legacy API. It does not implement the API, it calls it. Alternative is implementing a number of wrappers each encapsulating a particular service of the API. Both support incremental replacement of legacy services.



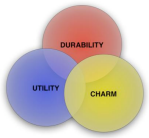
Legacy Wrapper - 2

- Consequences - depends on proper functioning and support of the legacy system(s) since they are doing the until they are replaced. Supports the philosophy of the pattern community, "an aggressive disregard for originality>"



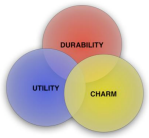
Proxy

- Pattern Name -proxy, aka surrogate, (similar to lazy evaluation)
- Problem - creating an object is expensive, yet you need to manipulate the object.
- Solution - create a stand-in that acts like the object but only actually creates the full blown object when necessary. There are different types: remote provides a local representative, virtual is a placeholder and creates it only on demand when absolutely needed, protection provides access according to access rights and



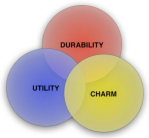
Proxy - 2

- Solution (cont'd) - smart reference is a smart pointer that can load a persistent object, reference count or maintain locks to manage access. Proxy is placed in front of the actual object and has the same interface. It controls access and can be responsible to create or delete it.
- Consequences - can hide the fact that a real object exists in a different address space. It can also do additional housekeeping duties.



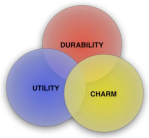
Chain of Responsibility

- Pattern Name - chain of responsibility
- Problem - object that provides services isn't known explicitly by requesting object because object servicing it depends on context and availability of resources, information, … need to present multiple objects that can potentially handle request
- Solution - handler defines an interface for handling requests and pointing to the chain of handlers (successor chain) that the request propagates through until an object handles request.
- Consequences - clearly reduces coupling of a sender to a particular receiver, but no guarantee that request will be handled (it falls off the chain). Adds flexibility in design of receivers



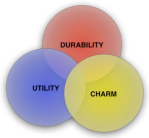
Command

- Pattern name: Command, a.k.a action, transaction
- The problem - lets toolkit objects make requests of unspecified app objects by turning the request into an object -- do not know receiver or operations it will do --- e.g. menu class
- The solution - bare bones is a class with interface for executing operations. For sequence of commands it is a for loop enclosing the execute.



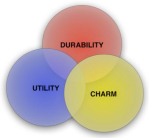
Command-2

- (cont-d) Uses:
  - Specify, queue and execute requests later-can have lifetime longer than original request
  - Can store state to support undo, create persistent log of changes for recovery
  - Support transactions
- The consequences
  - Separates request from actual command (object that knows how to do request)
  - Can manipulate commands, so therefore you can create a macro-command that does a sequence of commands
  - New commands are just added
  - Issue of how intelligent a command should be
  - Beware of side affects in undo/redo



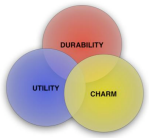
Interpreter

- Pattern Name - interpreter
- Problem - need to define a grammar for simple languages that would be frequently used. This includes representing sentences and interpreting these sentences. Need to represent a BNF. The grammar must be fairly simple and there must not be efficiency concerns.
- Solution - define a syntax tree of terminal and non terminal expressions. Terminal expressions implement specified operator, non terminal calls itself recursively until reaching a terminal expression. Context is also passed for information (such as variable bindings) that are global to the interpreter.



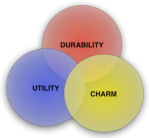
Interpreter - 2

- Consequences - easy to implement, change and extend grammar. Larger grammars are hard to maintain. Can add new ways to interpret the expression for type-checking, pretty-printing …



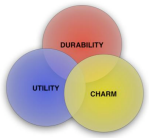
Iterator

- Pattern Name - iterator
- Problem - need to sequentially access elements of an aggregate object (e.g., object representing a list) without exposing underlying representation. Provides a uniform interface for traversing different aggregate structures.
- Solution - define interface for accessing list's elements and keeping track of current element, implementing first(), next(), CurrentItem() and IsDone().
- Consequences - supports variations of traversal algorithm, simplifies the aggregate since iteration is defined outside the aggregate. Because of this more than one traversal can be in progress on the same aggregate object.



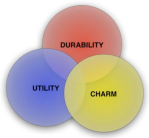
Mediator

- Pattern name: mediator
- The problem: OO encourages distributing behavior among objects but it can result in a lot of highly interconnected objects that become monolithic and hard to change (high coupling), common and not a good thing! Also since responsibilities are distributed, behavior can be difficult to change because it is distributed among so many objects.
- The solution - the mediator object controls and coordinates object interactions, keeping objects from referring to each other and providing developer/designer with a single place to change things if interaction changes. Acts as a communication hub for the objects.



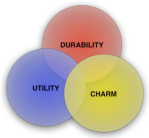
Mediator-2

- The solution (cont'd) uses:
  - Objects communicate in well-defined but complex ways
  - When reuse of an object is hindered because it refers to and communicates with many objects
  - Behavior used by several classes should be tweakable w/o subclassing
- The consequences:
  - Limits subclassing by localizing behavior that can be distributed across objects
  - Promotes loose coupling
  - Changes many to many to one to many
  - Separates object interaction from their behavior
  - Mediator can become a monolith that is hard to maintain -- complexity has to go somewhere



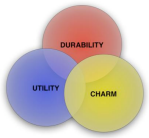
Memento

- Pattern name: Memento a.k.a as token (definition a keepsake)
- The problem: in cases where it is necessary to record internal state of an object (fault tolerance), e.g., undo, rollback state. Objects encapsulate the state that needs to be saved and exposing it would violate encapsulation. Also useful for reflection.
- The solution: stores a snapshot of the original object, the memento's originator that initializes the memento. Only the originator can store and retrieve, thus preserving encapsulation.



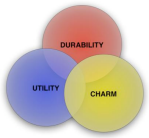
Memento-2

- The solution (cont'd) Uses:
  - Snapshot of all or portion of an object should be saved to restore later
  - Direct interface would expose implementation details and break encapsulation
- The consequences:
  - Preserves encapsulation
  - Simplifies originator … does not need to store the state
  - May be expensive if tons of state must be saved or if the originator needs to checkpoint often due to its frequent use
  - Impacts caretaker which garbage collects



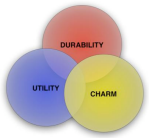
Observer

- 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.



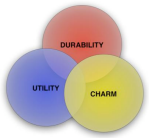
Observer - 2

- 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.



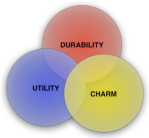
State

- Pattern Name - state
- Problem - need to alter an objects behavior when its state changes. Need to establish different operational modes for each state. For example an object representing a phone call.
- Solution - implement subclasses that provide state specific behavior.
- Consequences - localizes state specific behavior and partitions behavior for different states, permitting new states to be added easily. Makes state transitions explicit and enables state objects to be shared and they act like flyweights.



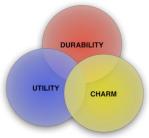
Strategy

- Pattern Name - strategy, aka policy
- Problem - different variants of an algorithm are needed, need to hide data an algorithm uses from clients.
- Solution - define classes for each strategy and an interface that is common to all supported strategies.
- Consequences - provides a way to represent hierarchies of related algorithms, provides a way to vary an algorithm supporting a class dynamically. Clients must be aware of the strategies and could increase number of objects unless you implement as stateless objects using a flyweight approach.



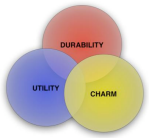
Implementing abstract classes

- Captures common behavior
- 3 kinds
  - Base method - behaviors generally useful to subclasses, implement in one place behavior used by all subclasses, e.g., error message for dividing by 0
  - Abstract methods - default behavior that subclasses are expected to override, placeholders, meant to specify classes responsibilities
  - Template methods - step by step algorithms, may include abstract methods that need to be defined, base methods, other template methods or a combo. Purpose is to abstract steps of an algorithm. First draw border then draw interior.



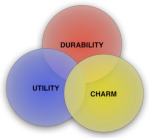
Template method

- Pattern name: template method
- The problem: define skeleton of an algorithm, deferring some definition to subclasses.
- The solution: template method defines an algorithm in terms of abstract operations that subclasses override. It fixes the ordering of the algorithm and the minimum that needs to be there.



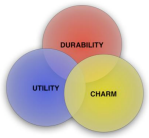
Template Method-2

- Solution (cont'd): Uses:
  - Implement the invariant parts of an algorithm
  - To localize common behavior among subclasses, generalizing
  - A fundamental technique for code reuse, especially in clas libraries
- The consequences:
  - The Hollywood principle - don't call us, we will call you. A parent class calls operations of subclass
  - Differentiate between operations that are hooks (may be overridden) and operations that are abstract and must be overridden



Visitor

- Pattern Name - visitor
- Problem - operations are required on the elements of an object without changing the classes of these elements and these elements are heterogeneous, requiring potentially different operations for each type. Also need to define additional operations on the elements, yet they rarely change.
- Solution - create a visitor for these elements and two hierarchies one for the elements acted on and one for the visitors that define operations on the elements. A new operation is created by adding a new subclass.



Visitor - 2

- Consequences - makes adding new operations easy, simplifies object class by using a visitor to gather related operations and separate unrelated rather than spreading it around original object structure. Visitors can also easily accommodate state in a well defined place. Adding new elements is hard since it requires new implementations in each visitor class. It also breaks encapsulation since an objects internal structure must be made available to the visitor