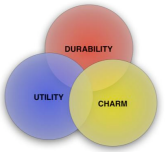
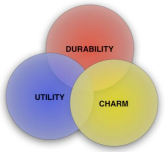


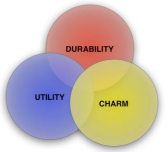
Class 7 CIS 573
(lecture 6)

Gregg Vesonder
University of Pennsylvania
Penn Engineering - Computer & Information Science



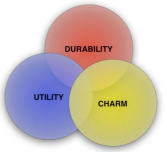
Roadmap

- Mid-Term
- Quality
- Coding
- Readings this class: Brooks chapter 12, Sommerville chapter 29, Andersson, et.al., chapter 7.
- Readings next class: Sommerville chapter 23
- Readings next week – posted on wiki Sunday, If not earlier



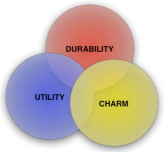
Critical Dates

- Every class project review
- ~~July 23rd Mid Term~~
- August 6th log books due
- August 11th project presentations
- August 13th Final



Teams

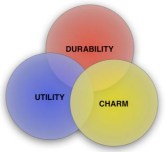
- Team 1 – Klein Keane, Beck, Buchman, Richardson, Nunez
- Team 2- Wilmarth, Caputo, Xiang, Francis, Nanda
- Team 3- Noronha, Fang, Huang
- Team 4-Whitehead, Liu, Ratnakar



Project Reports

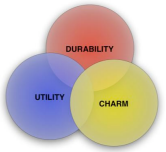
- Presentation each class
  - Green, yellow, red –simplified model + gaps
  - Current pressing issues
  - What was done since last class
  - What will be done before next class
  - Gaps

Penn CIS



Log Book

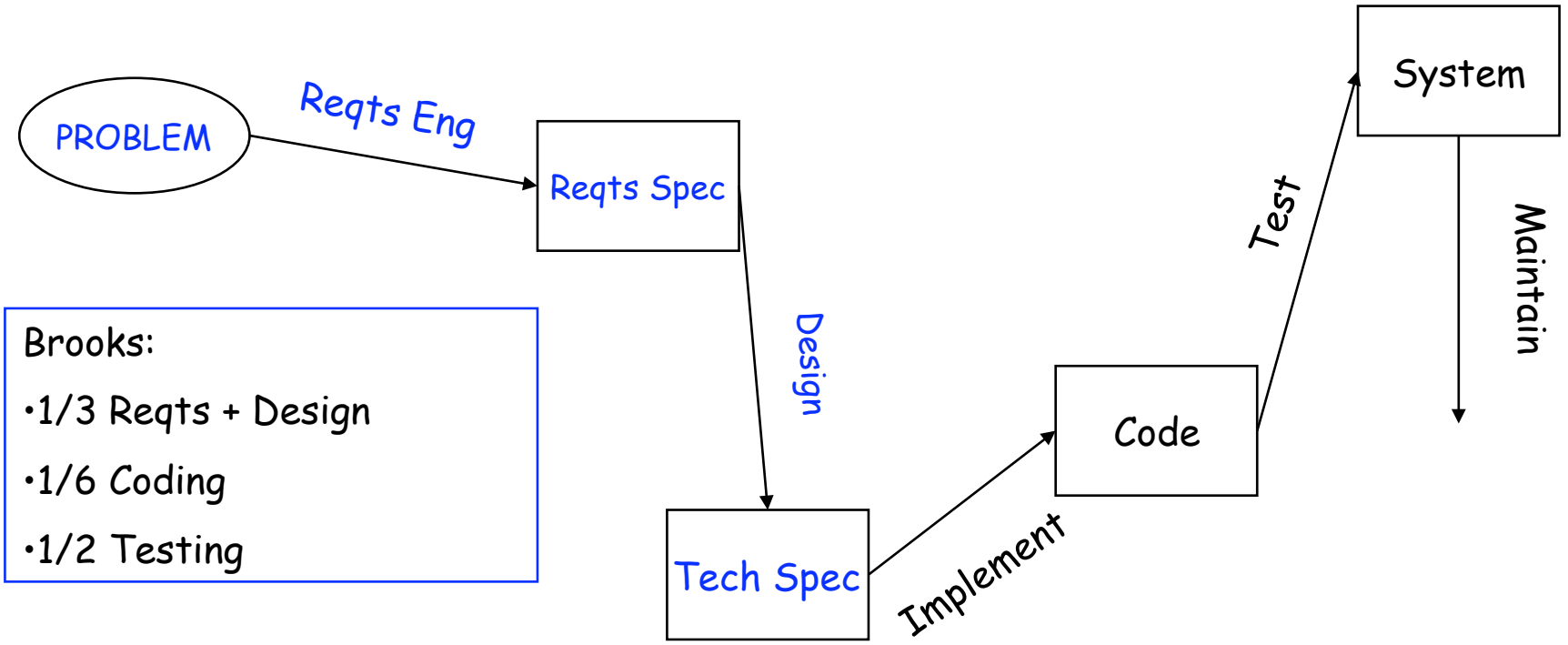
- Star (language) Wars

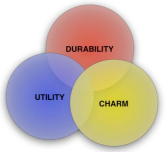


Software Engineering Knowledge

- SWEBOK, SoftWare Engineering Body Of Knowledge:
  - Software requirements analysis
  - Software design
  - Software construction
  - Software testing
  - Software maintenance
  - Software configuration management
  - Software quality analysis
  - Software engineering management
  - Software engineering infrastructure
  - Software engineering process

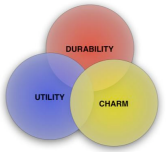
Simplified Model





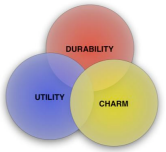
QAW

- Quality Attributes Workshop - facilitated method engaging stakeholders early to discover driving Quality attributes of a software intensive system
  - Results in creation of prioritized and refined scenarios
  - Provides description of Quality requirements before architecture is developed
  - It is system centric and stakeholder focused, done before software architecture is created
- Critical Quality attribute must be articulated and well understood early so it influences architecture - move to the left!
- Quality attribute examples: security, reliability, modifiability, performance, interoperability, portability



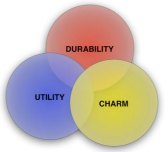
View of Traditional System Development

- Operational Descriptions
- High Lev Functional Requirements
  - Legacy Systems
  - New Systems
- A Miracle Occurs - Quality attributes are often missing from requirements document or, at best, vaguely understood and described
- Specific System Architecture
- Software Architecture
- Detailed design
- Implementation



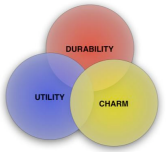
QAW-2

- Motivation is to not rely on the miracle but clearly articulate what is needed by scenarios describing the stimulus, describes agent or factor that initiates system to react and a response, the systems reaction to the stimulus. Including the environment, the context (e.g. peak load, normal operation, maintenance mode).



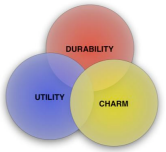
Steps of QAW

- Step 1 - QAW presentation - description and participant introduction
  - 5 to 30 stakeholders
    - Stakeholders = usual suspects = end users, installers, administrators, trainers, architects, system and software engineers.
- Step 2 - Stakeholders present systems business/mission context, including high level requirements, constraints and identified Quality attributes
- Step 3 - Architecture Plan presentation: notional, preliminary architecture describing how requirements will be satisfied, key technical requirements and constraints and description of the system environment



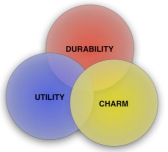
Steps of QAW-2

- Step 4 - Identification of Architectural Drivers - facilitators summarize these from presentations in Steps 2&3, ask stakeholders for clarification, additions & deletions. Results in final list of Quality attributes to drive scenario stage.
- Step 5 - Scenario Brainstorming, stakeholders generate scenarios, each stakeholder contributes 2, facilitators assure at least one scenario for each Quality attribute driver of Step 4. The Quality attributes are operationally defined by these scenarios, hopefully, avoiding ambiguity of vocabulary.



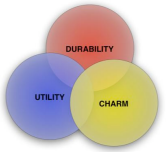
Steps of QAW-3

- Step 6 - Scenario Consolidation, similar scenarios are consolidated
- Step 7 - Scenario Prioritization, each stakeholder has N votes (N = 30% of # of scenarios), 2 passes, 1/2 of votes on each pass
- Step 8 - Scenario Refinement, work hard on clarifying descriptions of highly rated scenarios, including describing business/mission goals affected by scenario and describing relevant Quality attributes



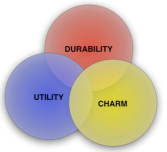
Example Scenario

- Scenario - when a garage door opener senses object in door's path, stops door in less than 1 msec.
- Business Goals - safest system; feature rich product
- Quality Attributes: safety, performance
- Stimulus - object in path of garage door
- Stimulus Source - object external to system, bicycle
- Environment - garage door is closing
- Artifact - system motion sensor & motion control software
- Response measure - 1 msec
- Questions - How large must an object be before detected
- Issues - train installers to prevent malfunctions



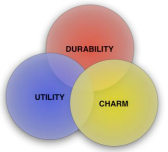
QAW Benefits

- Increased stakeholder communication
- Informed basis for architectural decisions
- Improved architectural documentation
- Support for analysis and testing throughout life of the system



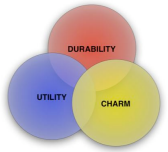
Metrics

- Project: metrics used specifically but not solely by management to control current projects and provide feedback for future projects
- Technical (Individual?): used by an engineers to improve their performance



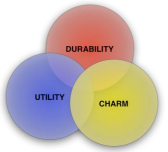
Technical/Individual Metrics

- Halstead
- McCabe
- Fan-in/Fan-out



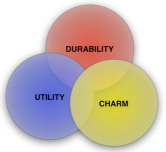
Halstead -"software science"

- Stresses syntactic units rather than LOC
- Model components:
  - Operators - actions: +, -, *, /, if-then-else,…
  - Operands - data: variables and constants
  - 4 basic entities (used in a bunch of equations)
    - n_1 - # of different operators
    - n_2 - # of different operands
    - N_1 - total occurrences of operators
    - N_2 - total occurrences of operands
  - Length of Program for Halstead: $N = N_1 + N_2$



Halstead uses

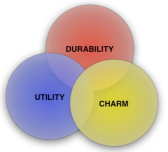
- Simple to calculate, no in-depth examination of structure
- Measure of overall quality of programs - simplicity/bloat criteria
- In conjunction with others, helpful in maintenance and initial programming
- Substantial literature
- At surface level requires completed code so not good in estimation but with certain assumptions N can be calculated early on
- Does not account for complexity of interfaces



McCabe's Cyclomatic Complexity

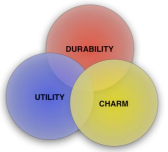
- Based on a directed graph showing control flow of program thereby showing the number of independent paths in the program
- Cyclomatic Complexity, CV= e - n + p + 1 where:
  - e = # of edges
  - n = # of nodes
  - p = # of connected components (1 for main program 1 for each procedure)
- 10 should be upper limit of complexity for a component according to McCabe

Eclipse metrics plugin



McCabe Uses

- Great for testing because it uncovers all linearly independent paths
- Does not add more complexity to nested loops and in general does not consider context
- Unlike Halstead does take into account control flow complexity, but not data therefore, often used together
- Useful for individual developer feedback and during maintenance

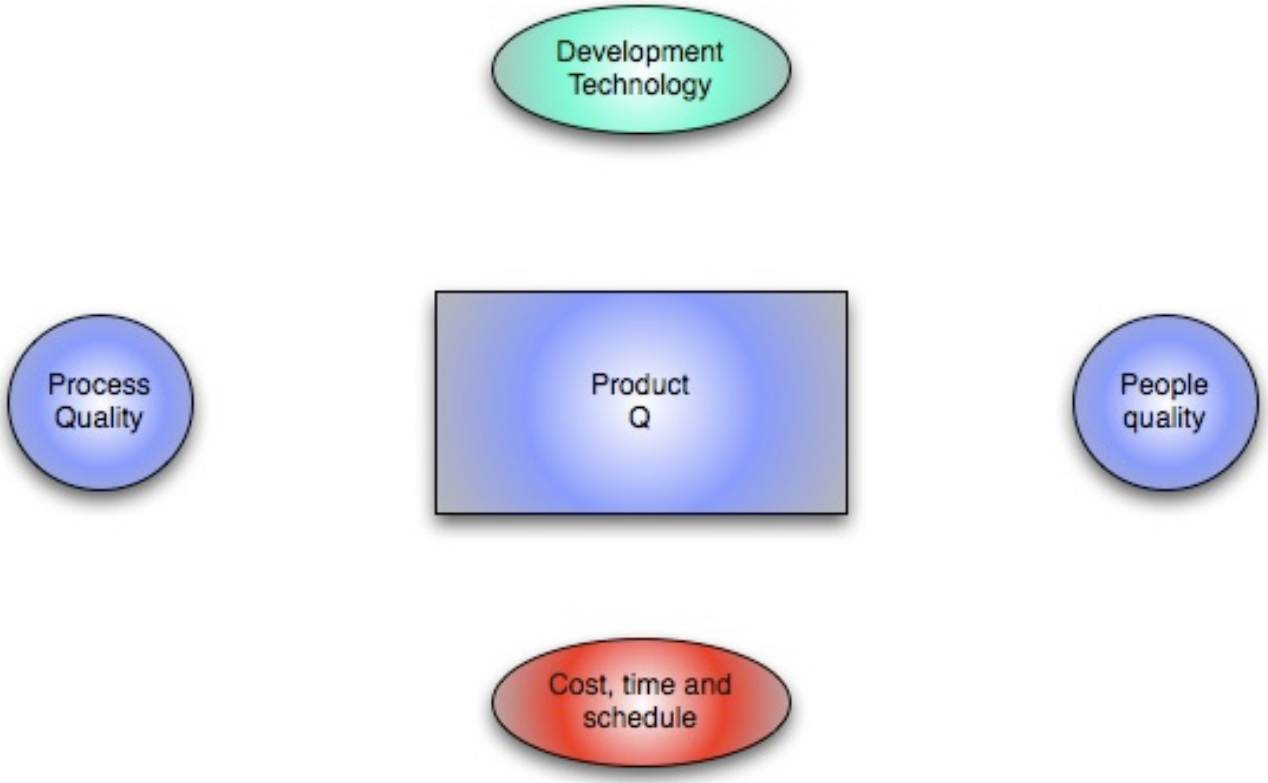


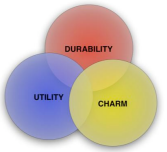
Fan In/Fan Out

- Measures interaction - basically, # of modules that call a given module and number of modules called by a given module.
- High degrees of fan-in/fan-out are undesirable
- Typical equations: [LOC|Cyclomatic complexity]* (#fan-in*#fan-out)^2
- Takes into account data driven programs but underestimates (of course) complexity for programs/modules with little interaction

Quality Universe

Sommerville p.667

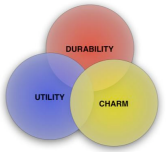




Reality Check

- The business is software, danger of a shift from developing software to developing processes, but …

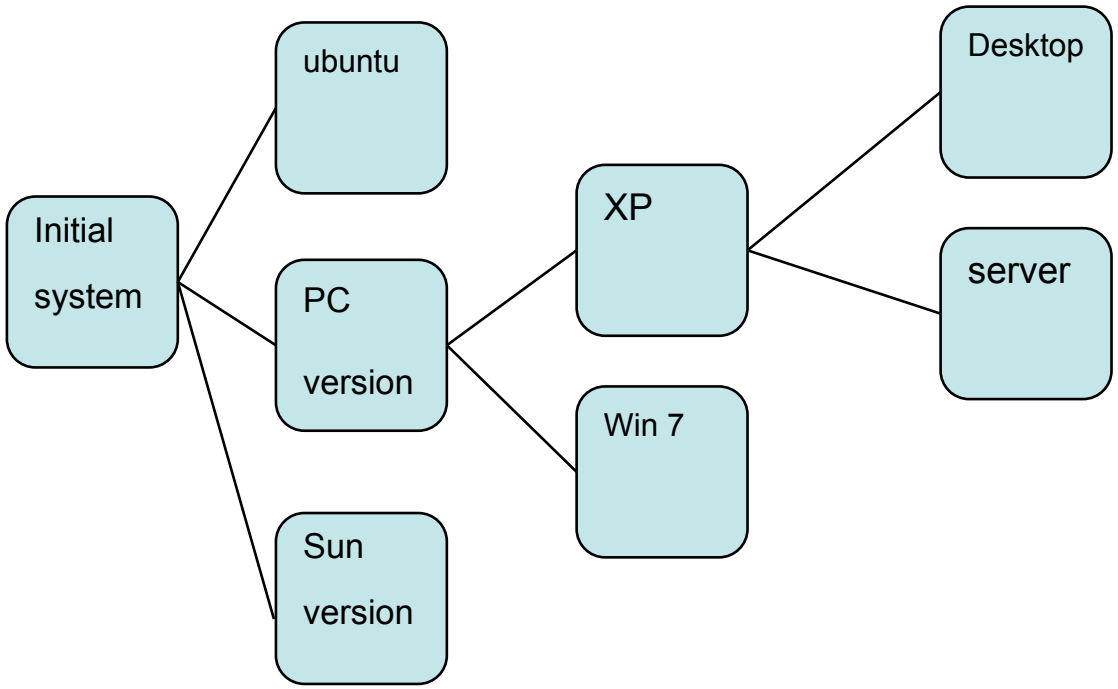
- Quality is recognizable

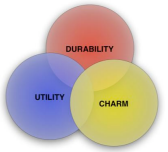


Configuration Management the problem

- Not a simple task!
  - Different versions of software usually is in the field during the life cycle
  - Different parts of the team are on different versions of the software and documents
  - The same release of a software product may have multiple versions consisting of different combinations of software components
- Configuration management is both a development and production issue

CM: Adapted from Sommerville (p.691)

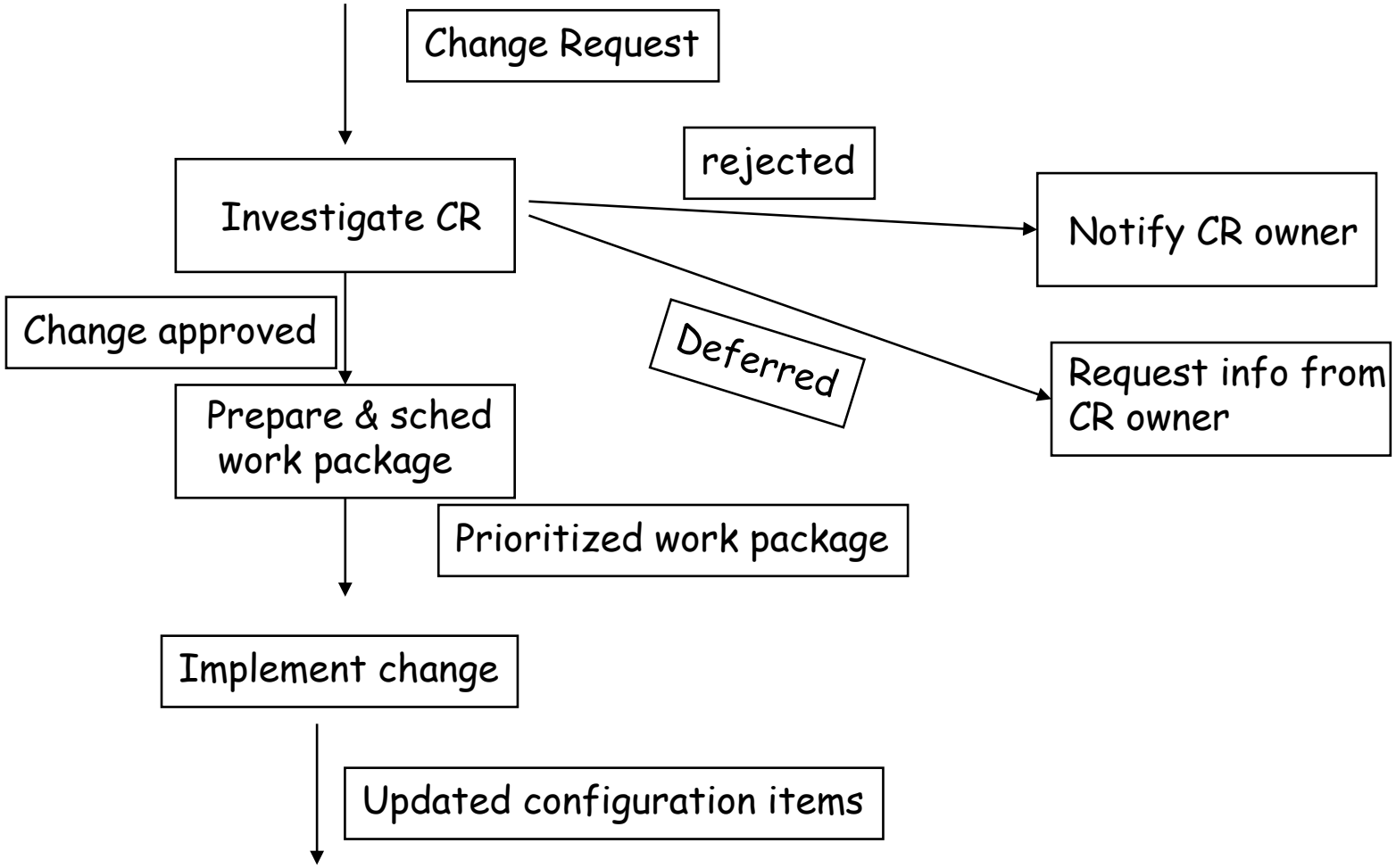


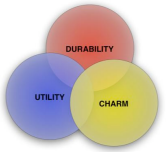


The Baseline

- IEEE - "reviewed and agreed upon basis for further development which can be changed only through formal control procedures"
- Contained in the baseline are configuration items: source, objects, requirements (p.75)
- Configuration management maintains integrity of these artifacts
- Major error- retrace steps through code, design documents and requirements specification -TRACEABILITY

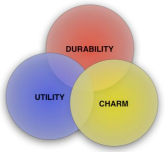
Workflow of CR (MR)





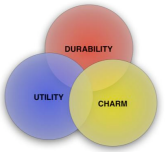
Configuration Management Tools

- Manage the workflow of CRs
- If item is to be changed, developer checks it out and item is locked to other users
- When item check back in revision history is stored
- All versions are recoverable
- Should be able to accommodate branching - necessary more times than you think!
- Configuration management tools are very sophisticated, keeps only the changes, the deltas and the remarks, timestamps and who did what - essential for Buildmeister and testers
- New tools are change oriented release configuration is identified by a baseline plus a set of changes.

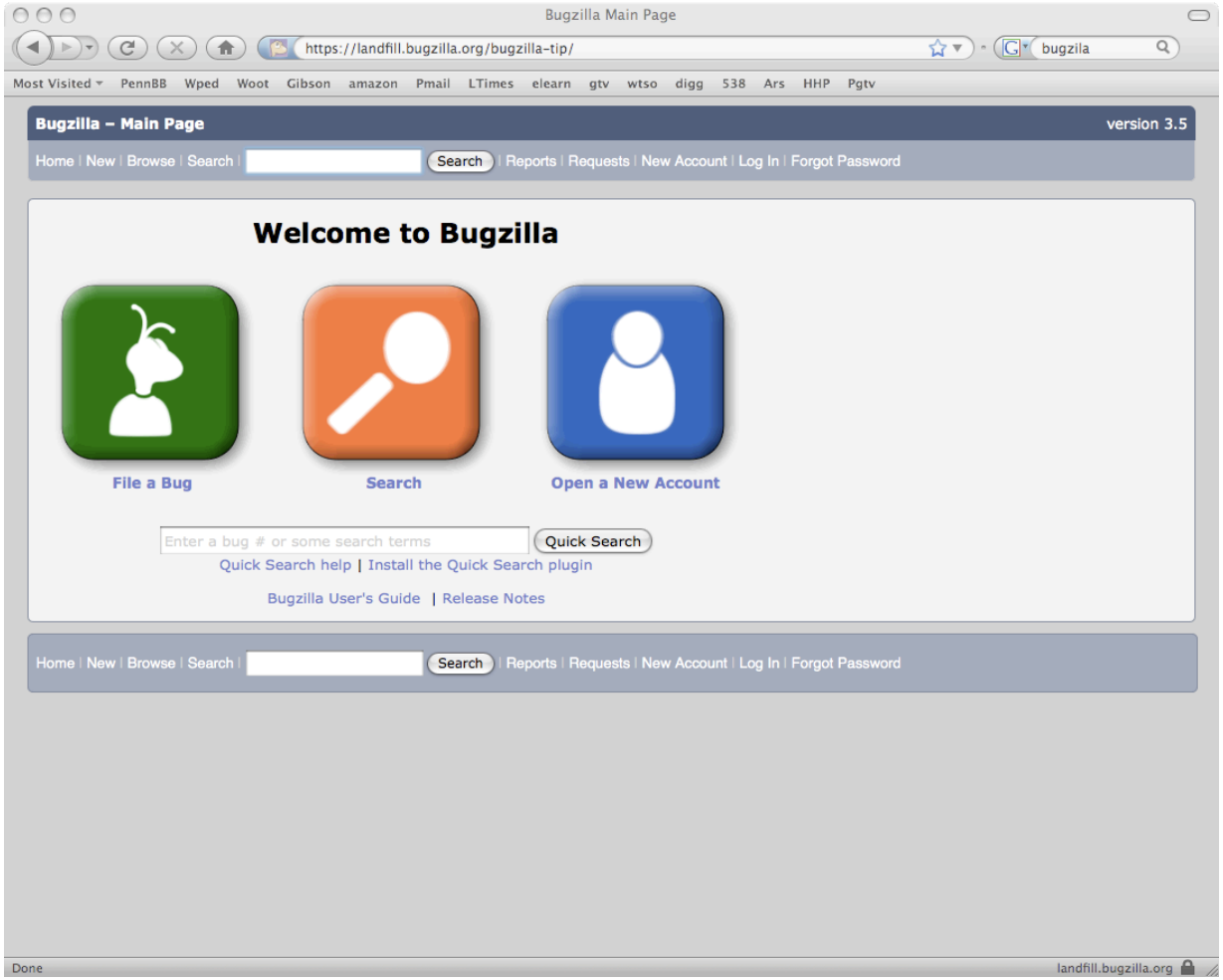


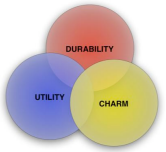
Configuration Management Plan

- Main parts:
  - Management: how project is organized and who has responsibilities related to configuration management. How are change requests handled?
  - Activities:
    - Who is on CCB, what are their responsibilities
    - What reports are required
    - What data is collected and archived - IMPORTANT


DURABILITY
UTILITY
CHARM

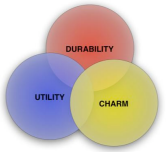
Bugzilla


Bugzilla Main Page
https://landfill.bugzilla.org/bugzilla-tip/
bugzila
Most Visited
PennBB
Wped
Woot
Gibson
amazon
Pmail
LTimes
elearn
gtv
wtso
digg
538
Ars
HHP
Pgtv
Bugzilla – Main Page
version 3.5
Home | New | Browse | Search |
Search
| Reports | Requests | New Account | Log In | Forgot Password
Welcome to Bugzilla
File a Bug
Search
Open a New Account
Enter a bug # or some search terms
Quick Search
Quick Search help | Install the Quick Search plugin
Bugzilla User's Guide | Release Notes
Home | New | Browse | Search |
Search
| Reports | Requests | New Account | Log In | Forgot Password
Done
landfill.bugzilla.org



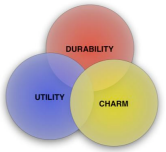
Game Development

- (but first a slight but appropriate digression)
- Roles of Game Development:
  - Producer - person responsible for managing people and processes responsible for the game
  - Game Designer - overall vision of the game and maintaining it
  - Level Designer - implements game using content creation tools created by programmers and assets generated by artists
  - Programmer - tool builder
  - Game Graphic Artist - know current context but be very broad
  - Much more "creative" based, developer as tool builder, amenable to software process factoring in this large difference
  - May (should) become more common



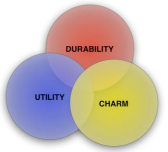
Micro Software Engineering

- About the developer
- SEI has the PSP, Personal Software Process, a bit about that next class when we discuss XP
- Derived from Hunt & Thomas, The Pragmatic programmer: From journeyman to master, Addison-Wesley, 1999.
- Supporting knowledge for agile and Open Source efforts



Axioms

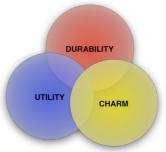
- Gets somewhat into the head of the hacker (good sense of word).
- Personal Aspirations:
  - Care about your craft
  - Sign your work
  - Think! About your work
  - Invest regularly in your knowledge portfolio
  - Don't think outside the box, find the box (does it have to be done this way?)
  - Gently exceed your users expectations
  - Organize teams around functionality; build teams like you build code



Axioms - Requirements

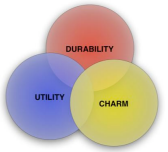
- Don't gather requirements, dig for them
- Work with a user to think like a user
- DRY - Don't Repeat Yourself (ambiguity)
- Keep knowledge in plain text
- Don't be a slave to formal methods
- Prototype to learn
- There are no final decisions
- Remember the big picture - MULTICS
- Use a project glossary
- English is just a programming language

Penn CIS



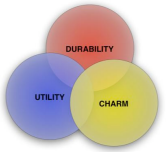
Axioms-Design

- Don't be a slave to formal methods
- Costly tools don't produce better designs
- Design to test
- Abstractions in code, details in metadata
  - Program for the general case put details elsewhere (no hard coding)
- Minimize coupling between modules
- Some things are better done than described



Axioms - Development

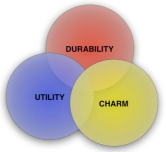
- Make it easy to reuse
- Eliminate effects among unrelated things (cohesion and coupling)
- Program close to the problem domain - design and code in your user's language
- Iterate schedule with the code
- Use a single editor well
- Use the power of command shells -GUI's do not always cut it
- Always use source code control
- You can't write perfect software - protect your users and your system
- Build documentation in, don't bolt on



Axioms-Test

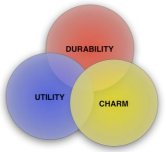
- Crash early - dead program is more instructive than one that limps along – hard failures
- Fix the problem, not the blame
- Test your software or your users will
- Test early, test often, test automatically
- Coding ain't done until all the tests run
- Use saboteur to test your testing - create your own mutants
- Find bugs once
- Refactor early and often

Penn CIS



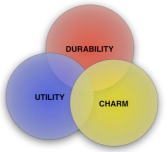
Refactoring

- Refactoring is changing software system without altering external behavior, yet improves its internal structure
  - Risky (Brooks up to .5 probability of introducing additional errors)
  - Disciplined - dangerous if not
  - Design focused, more improving design than code tweaking
  - Reversing entropy – anti-regressive maintenance
  - Extreme programming - continuous design
  - Design patterns as targets for refactoring



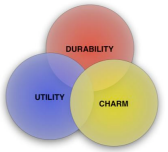
Begin Refactoring

- Need solid suite of tests - make tests self checking
  - Hand code examples and then compare results in the test
  - Great tests are easy to run and diagnose - you will be using them a lot - or should be
  - Most experienced developers worship tests and testers (or should!)
  - see any parallels to XP here?
- Break code into smaller pieces (extract method technique, later)
  - Look for variables in the fragment and their scope
    - Local variables, non modified, pass as parameter, modified return(so long as there is only one - recall java)
- Strive for small changes - since each change is small, errors easier to find



Refactoring Intro-2

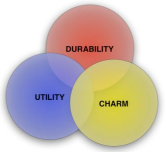
- P15 - "any fool can write code that a computer can understand. Good programmers write code that humans can understand."
- Rename variables for clarity and then test again!
- A method should be on the object whose data it uses (strong cohesion, weak coupling) - move method technique (also rename it to be understandable in its new context). And, of course test.
- Find any reference to the old method and adjust reference to new method (why two steps? Because you are striving for small, careful steps)
- Remove old method … test
- In summary move/rename, fix references, remove old, testing at each step.
- Redundant variables? Replace temp with query. Test
  - Discourages long, complex methods



Refactoring Intro -3

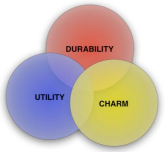
- Use refactoring to optimize clarity, increase cohesion and loosen coupling. This sometimes requires adding code and additional loops. You cannot tell if it affects performance until the whole process is complete. Unlike frequent testing, performance tends to be holistic and it is best to wait.
  - It is also "apples to oranges," if you add functionality and flexibility for future expansions of the application - then the performance hit is balanced by expandability and comprehensibility.

Penn CIS



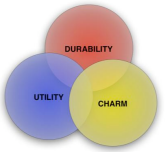
Principles in Refactoring

- Refactoring (noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- Refactor (verb) to restructure software by applying a series of refactorings without changing its observable behavior.
- Cleaning up code in an efficient, controlled manner
- Purpose is to produce code (and designs) that is/are easier to understand and modify - contrast with performance optimization which may make code harder to read (e.g., push to assembler)
- Does not change behavior - therefore should not change tests save for cases you missed



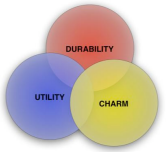
Principles-2

- Developing software involves refactoring … Switch from adding function (and tests) to refactoring .
- Refactoring:
  - Improves design of software (or at least preserves design of software)
  - Makes software easier to understand
    - Not always thought of when you are trying to get code to work
    - Also a way of understanding the code - (careful here) if you test it and functionality is preserved you prove that you understand the code.
  - Helps you find bugs - clarity of code highlights them
  - "Helps you program faster, good design = rapid software development"



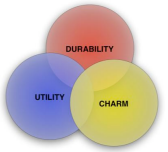
Principles - 3

- When to refactor:
  - When you are consistently wrestling with a design, for example consider the rule of 3 - the third time you do something similar, refactor
  - When you add a function
    - Refactor for understanding
    - Facilitates adding feature
  - When you need to fix a bug. This probably indicates a need to refactor since the software was not clear enough to see the bug initially
  - When you do a code review
    - A form of active reviews, you review by refactoring which aids in understanding
    - Keep such reviews small: one reviewer and the original author (XP)



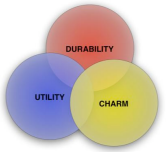
Beck-Difficult code

- Programs that are hard to read are hard to modify
- Programs that have duplicated logic are hard to modify
- Programs that require additional behavior that requires you to change running code are hard to modify - such changes may endanger existing behavior
- Programs with complex conditional logic are hard to modify - strive for simplicity



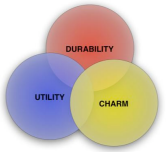
Principles - 4

- Convincing the luddite manager
  - Don't assume - other pressures
  - Active, more effective reviews
  - Quality
  - Schedule driven
  - Subversive



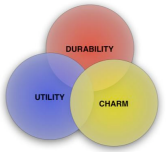
Issues with Refactoring

- Databases
  - Add an intermediate layer to isolate object (model) layer from database (model) layer
- Changing Interfaces
  - Refactoring does change the interface, e.g., rename method
  - Published vs. public interface - can't find and change all code that accesses it. For published you have to retain old and new interfaces until users can react to the change.
    - in java use deprecation facility to mark code as deprecated
    - Published interfaces are used too much? Makes refactoring difficult, change code ownership policy



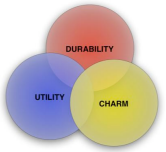
Issues with Refactoring - 2

- Design changes difficult to refactor - central issues that may force you to redesign rather than refactor (actually refactoring at a higher level as we saw with Evans)
- When not to refactor
  - Rewrite from scratch instead (see design) - current code simply does not work
  - Dice a large software component into small components with strong encapsulation, then refactor or rebuild decisions are made a bit at a time for each of the now smaller components. (actually you already made refactor decision)
  - When you are very close to a deadline - then you go into "debt"



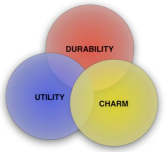
Refactoring and Design

- "With design I can think very fast, but my thinking is full of little holes" - Alistair Cockburn
- Programming is NOT a mechanical process
- Refactoring as the design process - XP
- Use CRC Cards - also may say a bit about size of systems
- Refactoring versus design, finding a reasonable solution rather than the solution (bit of a straw person)
- Simple vs. flexible (implies complicated design) - you only add flexibility where you need it



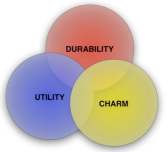
Refactoring and Performance

- Refactoring may make software run more slowly but will also make it more amenable to tuning.
  - Time budgeting - real time systems
  - Constant attention - everyone, everywhere concentrates on performance - narrow perspective - only small part of code usually affects performance -PROFILING
  - Refactor and use performance optimization stage late in the process, profile and tune in small steps
    - Smaller parts of refactoring add to finer granularity potentially adding to finer tuning
    - Slow first - much faster later



Bad Smells in Code!

- Certain structures in code suggest refactoring - the rest of the chapter discusses these indicators (the book implies ordering):
- See smells/refactoring table on back inside cover of book
- Duplicated code is one category of bad smells, some types:
  - Same expressions in 2 methods of same class - Extract Methods (110-pg number)
  - Same expressions in 2 sibling subclasses -Extract Method (110) then Pull-Up Method(322) -- other subtleties if not exactly the same.
  - Duplicated code in unrelated classes - extract class(149) or perhaps code belongs in only one method in only one of the classes.



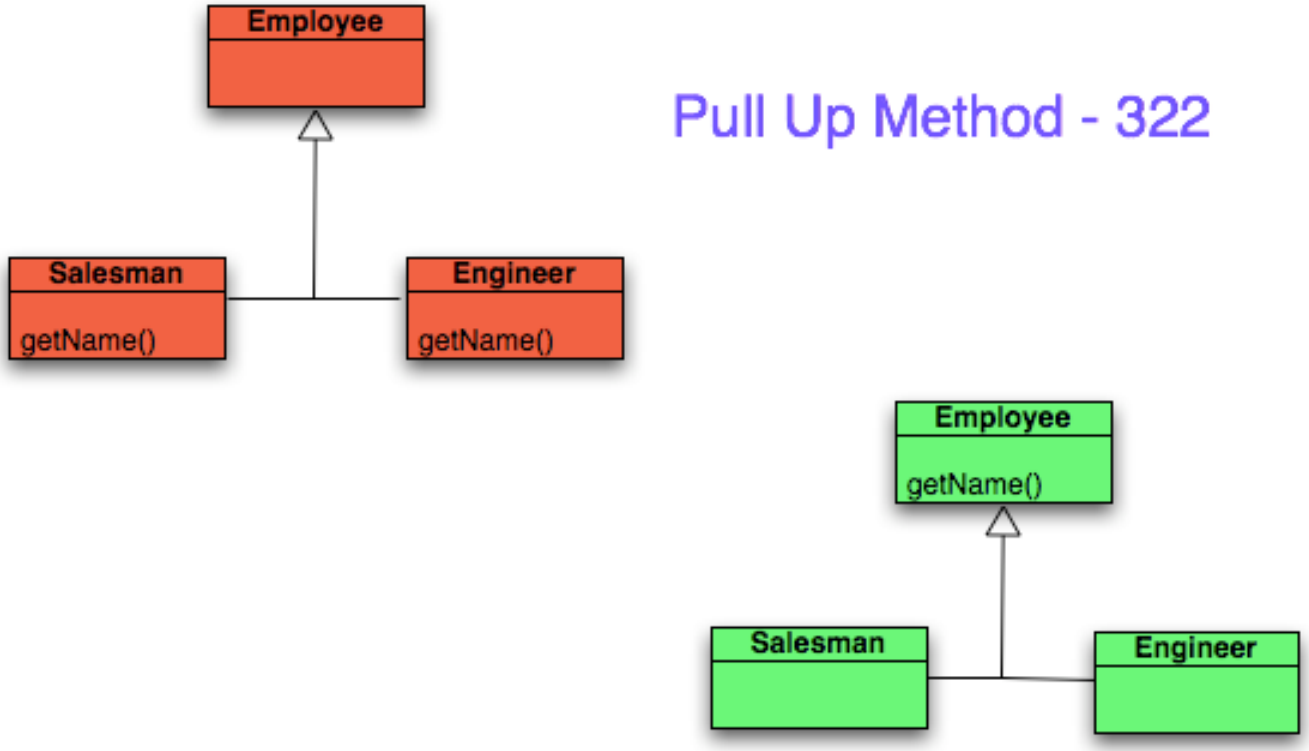
Extract Method (110)

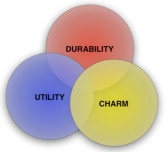
```
void printOwing(double amount) {
  printBanner();
  System.out.println("name: "+ _name);
  System.out.println("amount: " + amount);
}
```

```
void printOwing(double amount){
  printBanner();
  printDetails(amount);
}

void printDetails (double amount){
  System.out.println("name: "+ _name);
  System.out.println("amount: " + amount);
}
```

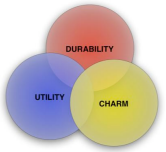
Pull Up Method





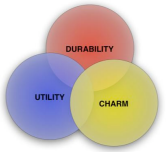
Long Method

- Long Method - OO metrics can detect this, the longer a procedure is the more difficult it is to understand, some types:
  - For typical long methods, use Extract Method (110), find parts of the larger method that go together and extract the method. (one key for clumps of code that can be extracted is descriptive naming)
  - Method with lots of parameters and temp variables -- attack this first
    - Replace temp with query (120)
    - Introduce parameter object(295)
    - Preserve whole object (288)
    - Still too many temps and parameters, replace method with method object (135)
  - Comments are a good indicator of clumps of code that can be replaced by a method
  - Conditionals and loops, sign for extraction - decompose conditional (238)



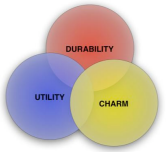
Large Class

- Large Class - one indicator is too many instance variables (suggests duplicated code)
  - Bundle a number of related variables and use extract class (149) or extract subclass (330)
  - Class with too much code is a breeding ground for duplicated code and chaos - eliminate redundancy in the class itself, tighten the code. Then extract class (149), extract subclass (330) and determine how clients hit code and extract interface (341) for each use.



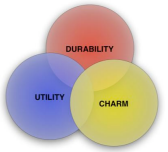
Long Parameter list

- Long Parameter List - once taught as a good thing, rather than using global data. In OO parameter lists are (should be) smaller.
  - Replace parameter with method (292) when you can get data in one parameter by making a request of a known object
  - Replace a bunch of data with the data itself using preserve whole object (288)
  - If you have several bits of data with no logical object, use introduce parameter object (295)



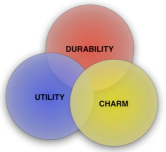
Feature Envy

- A method that is interested in a class other than the one it is in.
  - Invoking a bunch of get methods on another class to do its thing - use move method (142) and if only part of the method suffers from it use extract method (110) then move method (142)
  - If extracting data from several classes, where should it move? Where it is extracting most data!



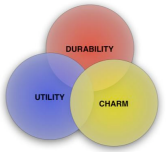
Data Clumps

- Bunches of data that hang together in different places should have their own object
  - Use extract class (149) to accomplish this turning clumps of data into an object
  - Then use introduce parameter object (295) or preserve whole object (288) to slim them down by shrinking parameter lists and simplifying method calling
  - Then look for cases of Feature Envy that would suggest behavior to move into the new class



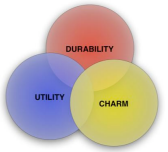
Inappropriate Intimacy

- Basically violating information hiding
  - Move method (142) and move field(146) separate pieces
  - Use extract class (149) if classes do have a common interest
  - Use Hide delegate (157) to have another class act as go betweens
  - Sometimes subclassing can contribute so replace inheritance with delegation (352)



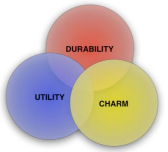
Popularity Poll

- High Runners: Move method (142), Extract Class (149), Inline class (154), Move field (146)
- Medium: Extract method (110), Introduce parameter object (295), Collapse hierarchy (344)
- Low: Rename Method (273), Preserve whole object (288), Replace inheritance with delegation (352), Hide delegate (157), replace type code with subclasses (223), replace type code with state/ strategy (227), introduce null object (260)



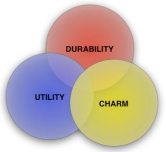
Building Tests

- As you might suspect, testing is important when refactoring
- Fixing a bug is easy, finding it is hard … substantial part of development is this activity
- Every method should have a test - make testing as painless as possible
- Test every time you compile
- Shades of XP, do the test before you do code, concentrates on interface, test should be so good/comprehensive that when the test works that indicates you are done coding



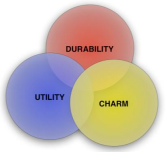
Testing-2

- Often when you refactor, you inherit a lot of code w/o self-testing
- The testing main on java - every class should have a main function that tests that class - can be tedious with lots of classes (but that too can be automated)
- Describes setting up a test environment using JUnit - open source
- Note that most of this is UNIT testing with some integration testing and system testing, especially if you are doing XP
- But if functional test/ users find something write a unit test(s) that exposes the bug.



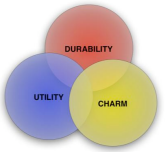
Testing-3

- Testing should be risk driven, focusing on key aspects, versus write a test for every public method (some are trivial)
  - The key is to make writing tests a doable chore
  - P101 "Don't let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs"
- In testing look for ways to try to break code
- Even check if exception handling works (groan)
- Inheritance and polymorphism makes testing harder since there are many combinations to test
- Build a good bug detector and run it frequently -- share info with your testing team and if you follow a regular framework it should be easy for them to test
  - As part of your development team standards, you should include a common format for unit testing



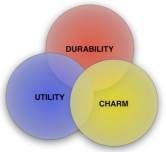
Refactoring Catalog Categories

- Composing Methods: extract method, inline method, …
- Moving Features between objects: move method, move field, …
- Organizing data: encapsulate field, replace data with object, …
- Simplifying conditional expressions: decompose conditional, consolidate conditional expression, …
- Making method calls simpler: rename method, add parameter, …
- Dealing with Generalization: pull up field, pull up method, …



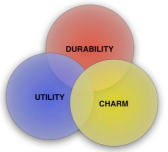
Composing Methods

- Composing methods - large bulk of refactoring, mostly methods that are too long
- Biggest issue with extract method (110) is local variables and temps



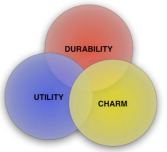
Extract Method

- Mechanics:
  - Create a new method and name it after the intention of the method
  - Copy extracted code from source method to new target method
  - Scan extracted method for vars that are local in scope to source method
  - See whether temp vars are used only w/in extracted code, if so make them temp vars of the new method
  - If these temp vars are modified by extracted code and if so see if you can treat it as a query and assign the result to the temp var concerned.
  - Pass into new method as parameters local scope vars that are read from the extracted code
  - Replace extracted code in source (original) with call to target method, You may be able to remove temp vars from original method, if now referenced solely by new method.
  - Compile and test



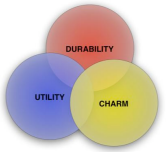
Last Words on these Refactorings

- Sometimes overall design is more appropriate
- The refactorings we just covered are on the whole locally oriented - The next few slides will cover refactorings larger in scope
  - One good aspect is that it makes metrics attractive
- You will become a better OO developer
- Judgement is still key



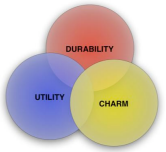
Big Refactorings - Chapter 12

- These are not as prescriptive
- They do not take hours - months - opportunistic redesign, do as much as you can when you can
- These are samples and not unsurprisingly the subject matter is dealing with inheritance and "OOness"



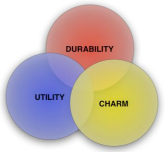
Tease Apart Inheritance

- Symptom: inheritance hierarchy doing 2 jobs at once -> turn into 2 hierarchies
- The trick is teasing apart these two jobs or dimensions the hierarchy currently represents - use graph paper to track the dimensions
- The problem is that this may be indicative of a larger design problem - gtv



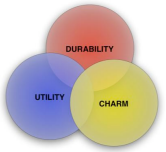
Convert Procedural Design to Objects

- Procedural -> turn data into objects, break up behavior
- Interesting process:
  - Take each record type and turn into dumb data object with accessors
  - Take all procedural code and put into single class
  - Take each long method and use Extract Method and others, then Move Method to appropriate dumb data class
  - Continue



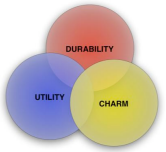
Separate Domain from Presentation

- Shades of MVC - separate the view/presentation from the domain logic, the model.
- Contrasts with 2-tier style where data sits in data base and domain logic sits in presentation classes
- Process:
  - Create domain class for each window
  - If a grid, create a class to represent rows
  - Examine data on window classes, if used only for presentation leave, else if not displayed, move to domain object else if displayed, create duplicate observed data so it is in both places
  - Examine logic in presentation move domain logic to domain classes
  - (a key is work from the window, what the user sees rather than working from DB, what the developer sees)



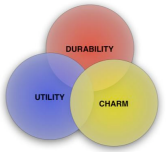
Extract Hierarchy

- Symptom: class doing too much work as indicated by conditional statements (Swiss-Army-knife class) -> create hierarchy of class with each subclass representing a special case
- Tease it apart and this may call for a much more top-down design process - however, and this is part of the attraction of refactorings, you may never have enough time to do an overall redesign and refactoring gets you to an "approximation" gradually.



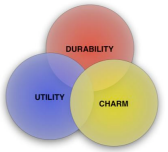
Refactoring Reuse Reality

- Refactoring should be seen as a middle ground
  - Makes design insights more explicit
  - Develops frameworks and extract reusable components
  - Clarifies software architecture
  - Prepares code to make additions easier
- Start with low level refactorings and only use a few of them
- Can be supported by automatic tools (because the refactorings are so low level)
- Brings the benefit of OO expertise in small steps - harder to get these things when you are not OO savvy.
- As refactoring becomes part of your routine it stops being considered (or feeling like) overhead to both you and your management.



Hueristics(Andersson, et.al.)

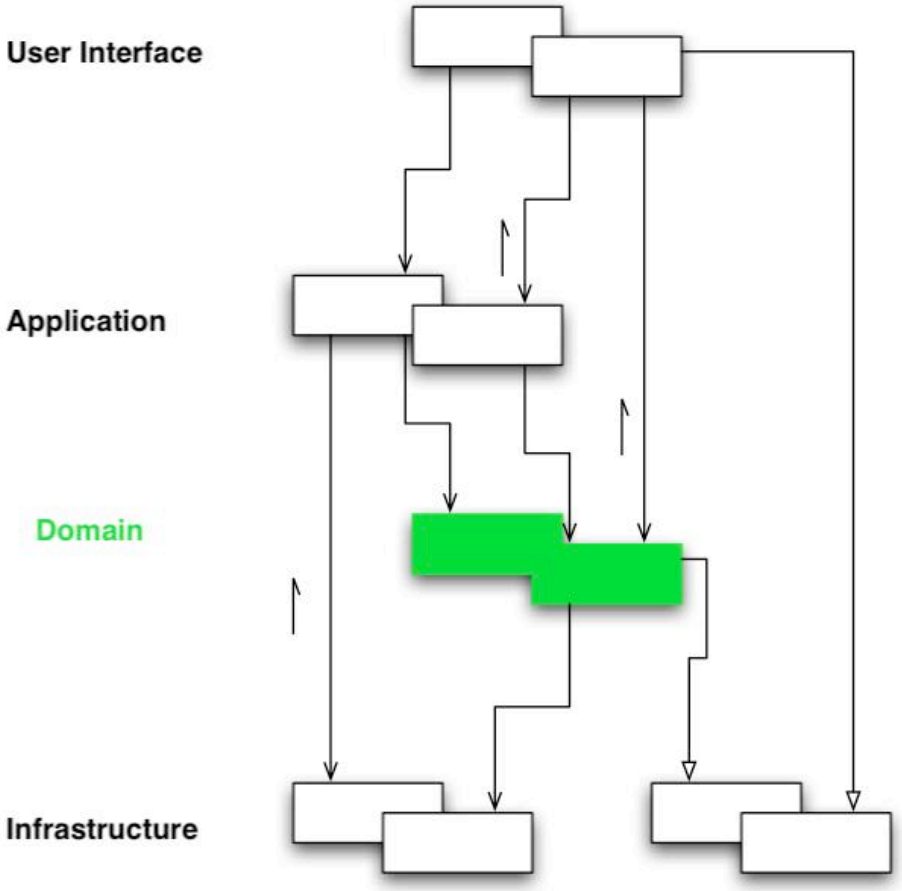
- Good developers may disagree on module division
- Roadmap
  - Where to find all software
  - Big picture – why was module built
  - Configuration information – meta data – no hard coding!
  - How to build
- Layers
- The "doc string" for every procedure what input it takes, what output it generates and how it does it
  - -need plain English to stitch it together!

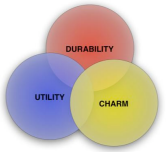


Building Blocks

- 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
  - The domain layer, not the application layer is responsible for fundamental business rules

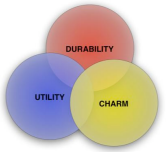
Layered Architecture





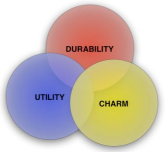
SERVICES -> Layers

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



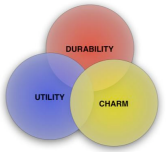
SUN – Coding Conventions

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html



Example Naming

Id type	Naming Rules	Examples
classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive.	class Raster; class Gene;
methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	getAccountData(); addUser()
variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. One character for throw away variables only.	int i; Float geneValue;
constants	The names of variables declared class constants should be all uppercase with words separated by underscores ("_")	static final int MAX_WIDTH;



References

- Futrell, Shafer & Shafer, Quality software project management, Prentice Hall, 2002, ISBN 0-13-091297-2
- Robertson, S. and Robertson, J., Mastering the requirements process, 1999, Addison-Wesley.
- Endres, A. and Rombach, D. A handbook of software and systems engineering. 2003, Addison-Wesley.
- Wirfs-Brock and Schwartz - http://www.wirfsbrock.com/pages/resources/pdf/the_art_of_writing_use_cases_slides_and_notes.pdf
- Fowler, M. Refactoring, Addison-Wesley, 1999.
- Others embedded in text