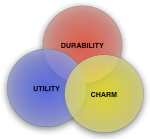


Class 11(lecture 10) SSW565

Gregg Vesonder
Stevens Institute of Technology

Roadmap

- Logbook
- Part 4 in Evans
- Security and arch/design
- Readings next class, chapters 1 thru 3 in Fowler, Refactoring

Key Dates

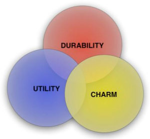
- Final July 20th – take home due July 23rd

Clarification

- Place your top talent in the core domain NOT the technical infra structure – sometimes easier said than done!

Logbook

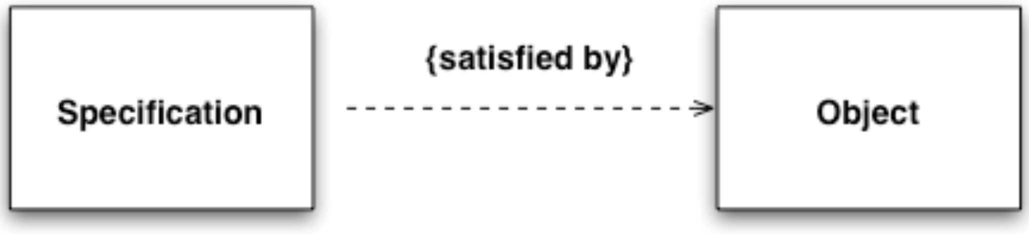
- "source code is free speech"
  - From a bumper sticker



Processes as Domain Objects

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

Specification



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

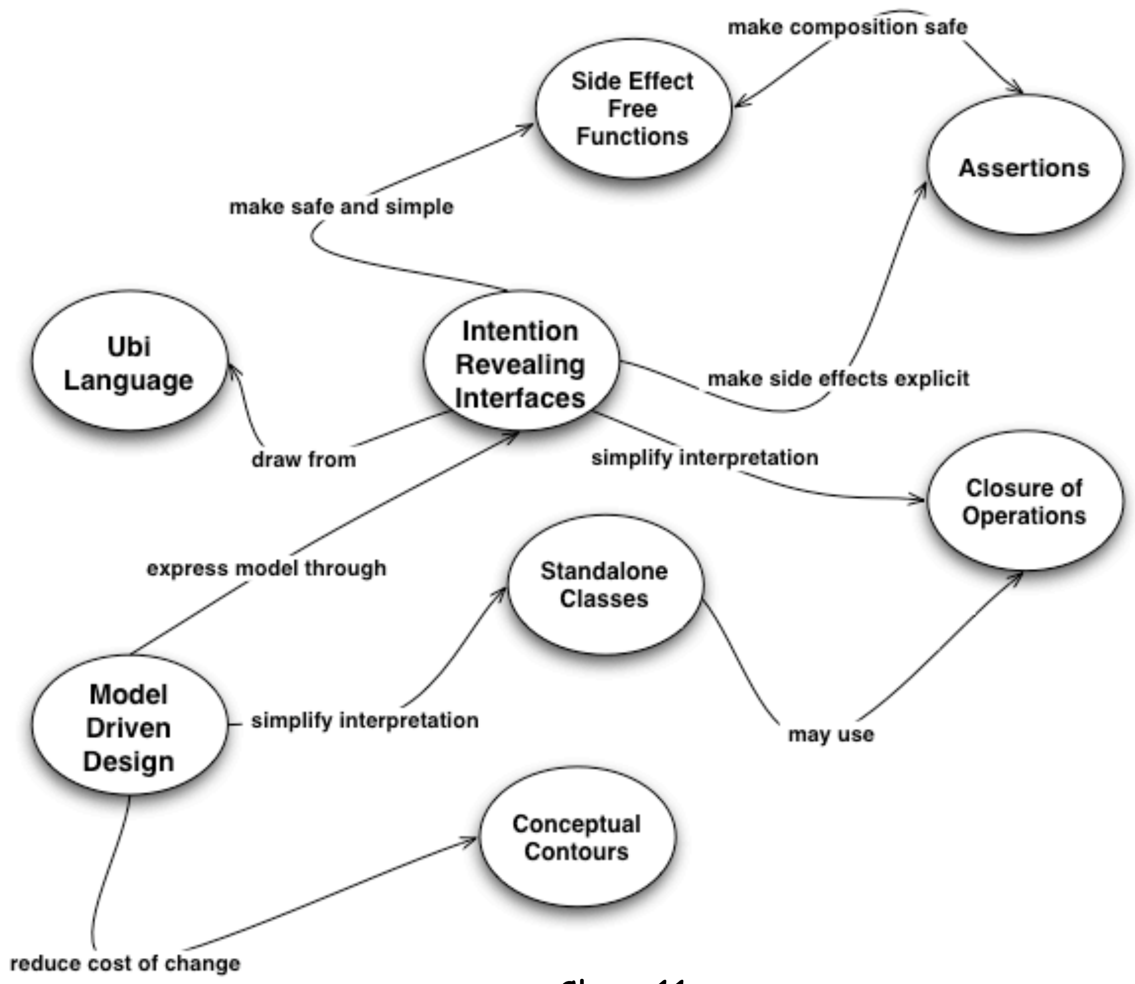
Use Specification

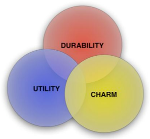
- To validate an object to see if it fulfills some need or is ready for some purpose (preconditions)
- To select an object from a collection (discovering overdue invoices)
- To specify the creation of an object to fit some need

Supple Design

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

Patterns and Supple Design





Supple Design - 2

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

Intention Revealing Interfaces

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

IRI-2

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

IRI - 3

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

IRI - 4

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

Side Effect Free Functions

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

Commands

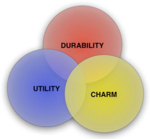
- On the other hand commands (aka modifiers) are operations that affect some change to the system--produce side effects.
  - Queries obtain information by accessing data in a variable possibly doing a calculation based on it
- Commands can not be avoided but can be tamed in 2 ways:
  - Keep commands and queries segregated, methods that cause changes should not return domain data: get_account vs deposit_into_account. Do all queries and calculations in methods that cause no observable side effects
    - Queries never cause state changes, what you do with them may
  - May be alternative models that do not call for existing object to be modified - instead create a new Value Object in answer to a query, hand off and forget
    - Value Objects are immutable after initialization, all their operations are functions

Assertions

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

Contracts

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

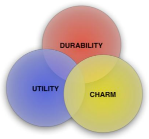


Conceptual Contours

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

In Summary

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

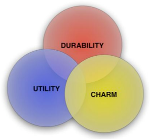


Standalone Classes

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

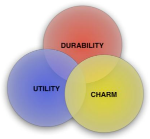
The Rest

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



How - Angles of Attack

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



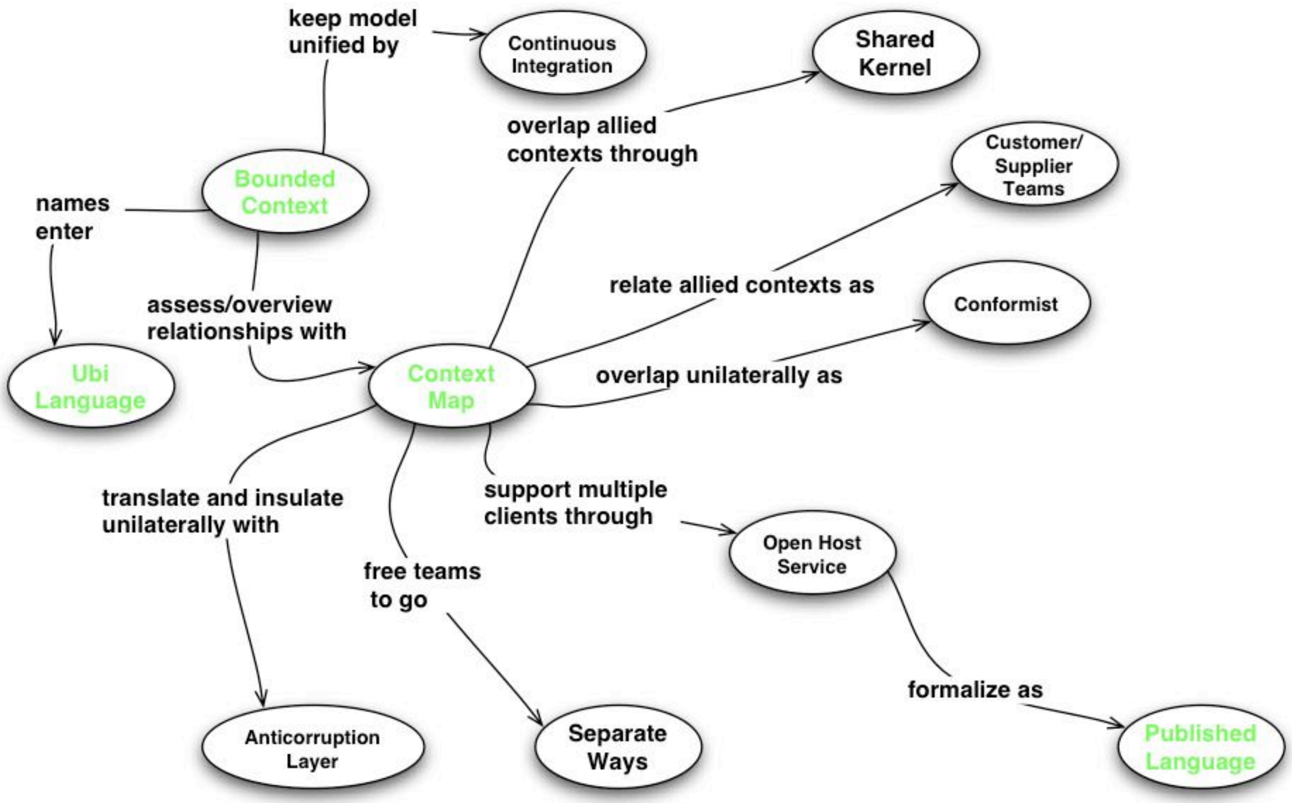
IV Strategic Design

- Time to Big! Small is beautiful, big is necessary!
- Three Broad Themes:
  - Context - a successful model of any size has to be logically consistent
  - Distillation - reduces the clutter (extraneous details) and focuses attention appropriately
  - Large-scale structure - in a very large, complex model one may not see forest for the trees, and therefore never achieve large scale structure
- Discussing architecture and design at higher levels in this section.

Context

- Unification - the internal consistency of a model such that each term is unambiguous and no rules contradict
  - Total unification for the whole model, especially on larger projects is not possible (because it is the real world)
- Focuses on techniques for recognizing, communicating and choosing the limits of a model and its relationships to others.
  - Bounded context defines the range of applicability for each model
  - Context map provides a global view of a project's contexts and relationships between them (aka big picture)
  - Continuous integration will keep the model unified once contexts are formed and mapped (update)

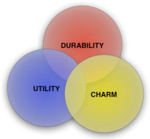
Model Integrity



A navigation map for model integrity patterns (14.1)

Multiple Models

- Large projects usually have several models. Key is that each model applies in a context (slice of code/team)
- Bounded context delimits applicability of a model so that team members have a clear and shared understanding of it and how it relates to other contexts
- In some sense it is a contract



Continuous Integration

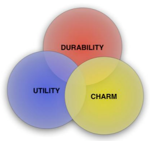
- Once you start breaking a system into modules it could encourage further segmentation
- Continuous integration protects against this reductionist tendency ensuring that all work within a context is merged and made consistent both in the model and the implementation
  - Step by step merge/build techniques
  - Automated test suites
  - Rules that set a time limit on un-integrated changes
  - Most Agile projects have daily integration
  - Scope is the bounded context

Context Map

- Overlap between project management and software design
- Name each bounded context and incorporate in Ubi language - therefore the bounded contexts should not be artificial barriers but pertinent to the domain
- Describe points of contact between models
- Map the terrain (bounded contexts and their relationships)
- Should be current and should be shared and understood by everyone on the project
- Contact points between bounded contexts are important to test
- Next explore patterns that could be used to relate models/ contexts.

Shared Kernel

- Designates a subset of the code teams agree to share (code or database design), e.g. infrastructure layer
- Elements should not be changed without agreement of all teams
- Test suites must reflect the necessary tests of all teams
- “trust, but verify”



Upstream/Downstream Issues

- Not good to be downstream - regardless of efforts, downstream system must be constantly vigilant
- Try to establish clear customer/supplier relationship
  - Jointly specified tests to validate the interface -- acceptance tests
- Usually only has a chance to work when you have shared management or shared concerns.
- However, some (most) times the downstream system is on its own -- conformist

Up/Down: Conformist

- In this scheme the downstream team makes do with what is sent downstream
- Close adherence to the upstream model (if you can get it)
- Make communication easy for upstream systems - go to their meetings
- But it can get worse!

Up/Down: Anticorruption Layer

- Usually a case with legacy systems that are being modernized (you do not know much of the existing model) or in a case when the upstream team(s) is/are uncooperative
- Create an isolating layer to buffer all the issues
  - Translates conceptual objects and actions from one to another
  - Wrappers are a variant - using a different protocol (rules of interaction) than that understood by the original code
  - Again test heavily to insure that things do not change
- Isolation strategies are expensive - make sure they are needed, maybe the projects should go separate ways

Up/Down: Separate Ways

- Declare a bounded context to have no connection to the other (upstream) contexts and find straightforward solutions within this context
- No sharing of logic and minimum (best is none) sharing of data
- Hard to remerge models such as these that have evolved in complete isolation - try other patterns to integrate some of it-- Open Host Service

Open Host Service

- For very popular sub systems that interact with a bunch of bounded contexts (popular services that many teams use)
- Define protocol that gives access to it as set of services
  - Use a one-of protocol for special cases (allow exceptions)
  - Test everything
  - Make sure there is shared model vocabulary in Ubi language or as
- Published language that is readily available to entire community -- XML is a great candidate and easy to use, stylized, HTML-like
- In the end having well - defined Bounded Contexts are the key, but what size?

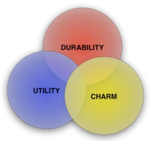
One or Many Bounded Contexts?

- One:
  - Flow between user tasks is smoother
  - Easier to understand one model
  - Translating between two models can be difficult
  - Shared language improves team communication
- Fewer than 10 team members

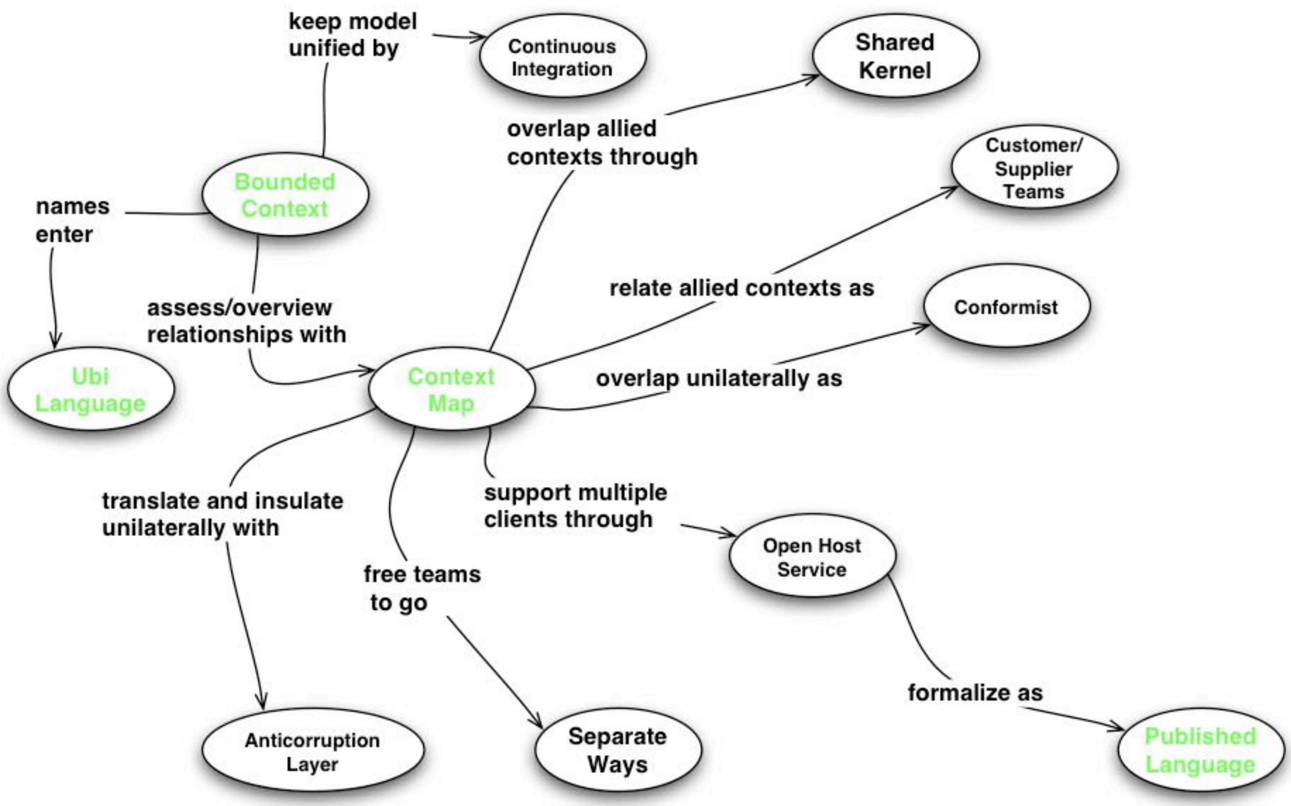
- Multiple
  - Communication overhead is reduced
  - Continuous integration easier with smaller teams and code bases
  - Larger contexts, greater skills
  - Encompass specialized jargons in separate contexts
- Each with fewer than 10 team members

Phasing Out Legacy Systems

- Incremental phasing
- Key is to decide on a testing strategy to insure sanity
- New and old run in parallel
- Iterate:
  - Identify functionality that can move into one of the newer systems
  - Identify modifications to anticorruption layer
  - Implement
  - Deploy
  - Identify any unnecessary parts of anticorruption layer
  - Consider deleting the superceded modules in legacy but be careful - ignoring them may be best



Model Integrity

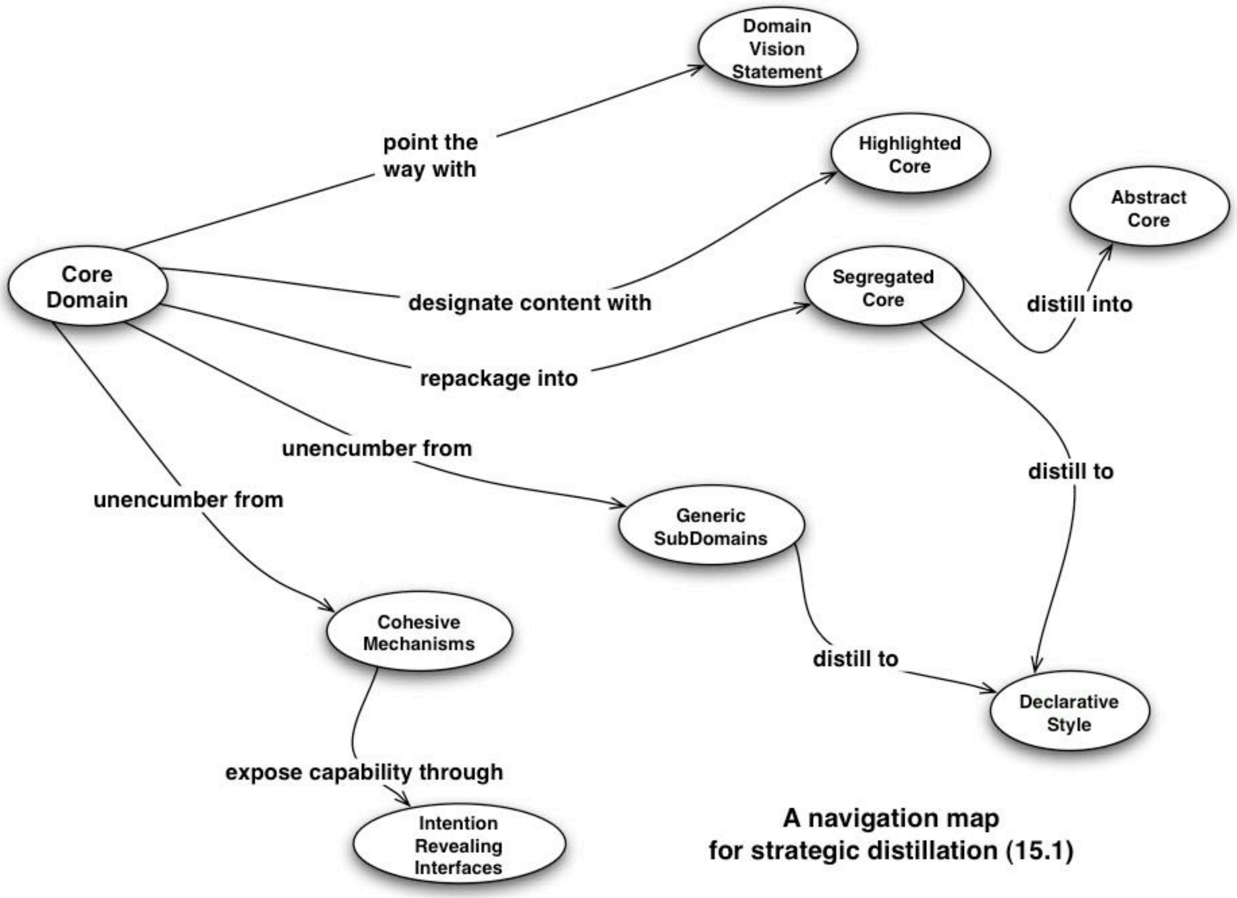


A navigation map for model integrity patterns (14.1)

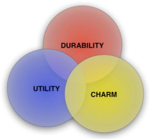
Distillation

- Tightening and condensing the model to the core knowledge necessary
- Advantages:
  - Easier to grasp the overall design
  - Concomitantly easier communication using Ubi language because it is simpler
  - Guides refactoring
  - Focuses work on key areas of model
  - Guides outsourcing, COTS component use and team assignments
- So let's open up the model a bit…

Distillation

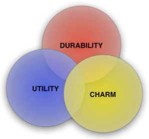


A navigation map
for strategic distillation (15.1)



Core Domain

- (Comments on Architecture are misplaced - in many ways the architect as conceived in this course would be an integral part of the model building)
- The specialized core is the part differentiating the application making it a business asset
- Should be the province of the most skilled developers, but sadly it often goes the other way
- Boil the model, find core domain, apply top talent to core domain
- Done by an escalation of distillations

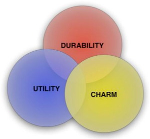


Generic Sub-domains

- Identify cohesive sub-domains that are not the motivation for your project
- Give them lower priority
- Consider off the shelf or published models (XML)
- Consider outsourcing - automated tests are the key (recurring motif)

Domain Vision Statement

- Similar to the architectural problem statement and this should constantly evolve
- Domain Vision Statement serves as a guidepost so that development heads in a consistent direction, test changes and additions against the statement
- Short and understandable, much less than 10 pages - broad terms
- Examples in book, p 416



Highlighted Core

- A few key diagrams or documents that provide anchor and entry points for the team
- Usually a separate document, distillation document but not the complete design document:
  - Must be maintained
  - Must be read
  - Must be useful, not another layer of complexity
- It should focus on those sections of requirements that present the essential, differentiating concepts that are the focus of the project
- When a code change affects the distillation document, it should be reviewed by the entire team, it is a key to trigger process, i.e., triggers review

Cohesive Mechanisms

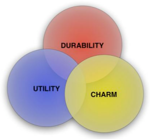
- Some algorithms sometimes dominate the core domain (tree example of organization chart tool) - partition into a separate framework with an intention revealing interface.
- Not a generic sub-domain because this is proprietary, essential to the uniqueness of the core domain
- One goal is to distill much of this so you can proceed to a declarative style, where modifications to the core can be scripted.

Segregated Core

- Refactor to separate supporting code and concepts, even partially understood concepts (as time goes on you can reincorporate if essential, recall that this evolves)
- Process for segregated core:
  - Identify core subdomain
  - Move related classes to new module - named for concept that relates them
  - Refactor code to sever functionality not related to concept
  - Tighten cohesion of module through refactoring
  - Continue with other core sub domains
- What remains in the Core Domain is the key focus of the application, the problem you are trying to solve

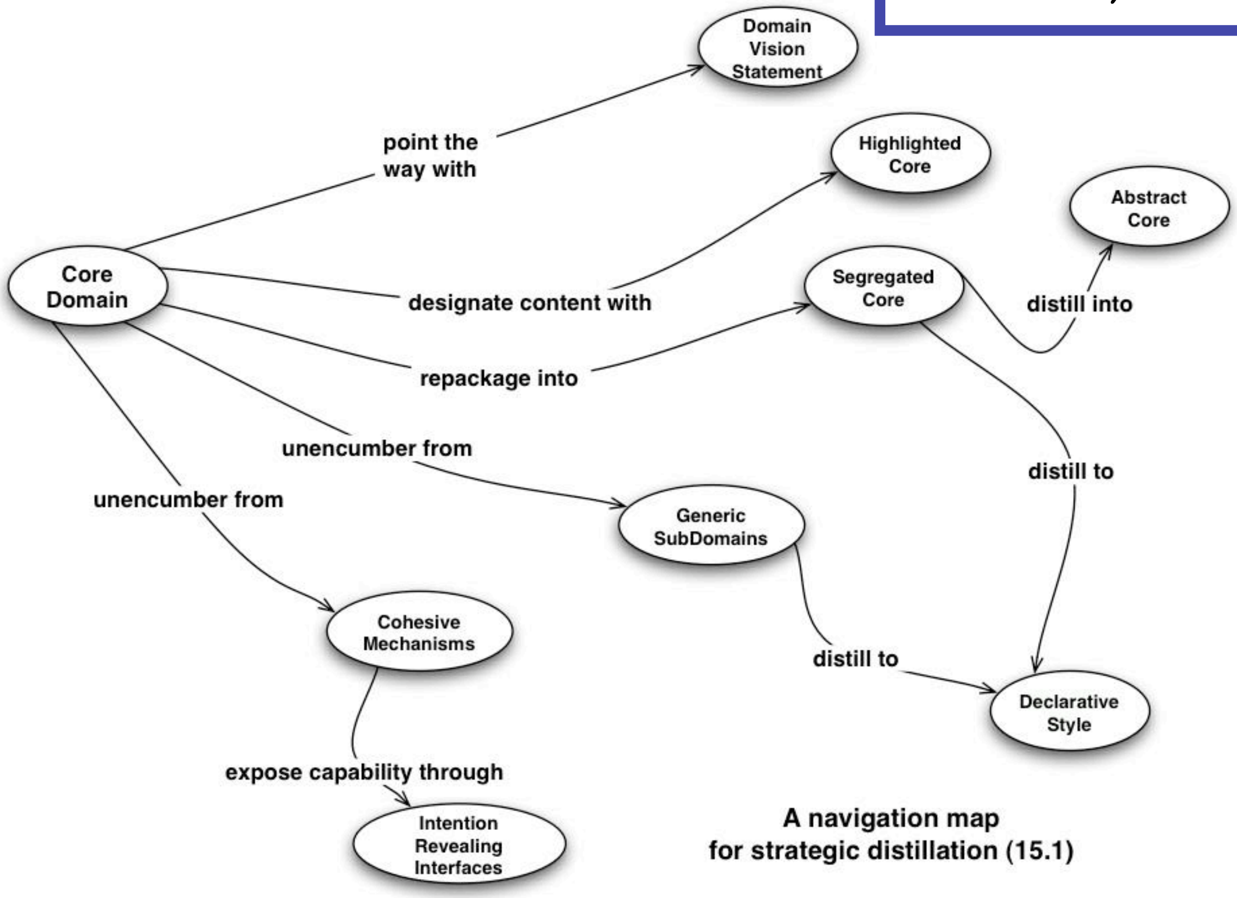
Abstract Core

- After all these efforts, core may still be too large for facile communication
- Therefore identify the most fundamental concepts of the model and factor into distinct classes, abstract classes and interfaces.
  - The goal is to have this abstract core reflected in code
- This cannot be done early -- requires a deep, mature understanding of the knowledge and is a reflection of the ongoing evolution of the architecture - result of multiple iterations



Distillation

Pulling it apart and getting to the essence, the conceptual integrity

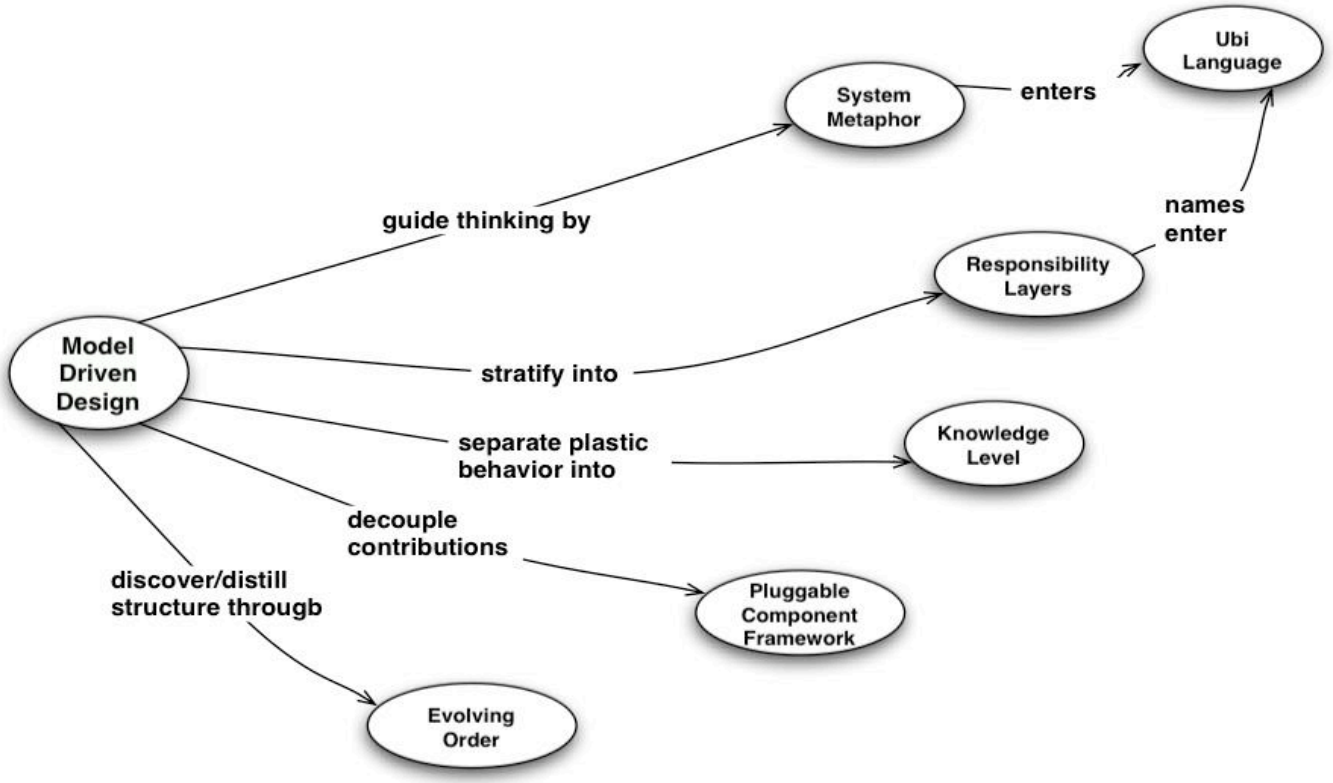


A navigation map for strategic distillation (15.1)

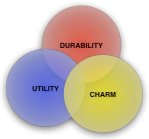
Large Scale Structure

- Lets you discuss system in broad strokes
- Avoids forest- trees syndrome, inhomogeneity of technical understanding syndrome
- A pattern of rules or roles and relationships that span the entire system and allow some understanding of each part's place in the whole -- without detailed knowledge of the parts responsibility
  - Understand the relationships in the system without understanding the details

Large Scale Structure



Large Scale Structure (16.1)



How simple can you make it?
How complex does it have to be?

Evolving Order

- Again our view of architecture is that it is adaptable, not a straitjacket
- The large scale structure evolves, just as the small scale structure
- ".. it is no mean feat to create a structure that gives the necessary freedom to developers while still averting chaos." (p.446).
  - The architectural challenge

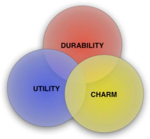
Simplicity is about subtracting the obvious and adding the meaningful

System Metaphor

- XP use of metaphor to bring order to a development project
- System metaphor is a loose, easily understood, large scale structure fitting into object paradigm.
  - They are difficult to find
- The desktop in modern interfaces

Responsibility Layers

- Layering of responsibilities
- Identify natural strata of the domain and use a relaxed layered system
  - Relaxed layered system allows components of a layer to access any lower layer, not just the one immediately below
- For example: operational, capability, decision support
- Choosing appropriate layers:
  - Should provide a unifying story ala games
  - Conceptual dependency, upper dependent on lower, lower stand alone
  - Conceptual contours - objects at different levels should have different rates of change



Factory Automation System

Decision	Analytical mechanisms	Very little state, so little change	Management analysis, optimize utilization, reduce cycle time
Policy	Strategies, constraints (based on business goals or laws)	Slow state change	Priority of products, recipes for parts
Operation	State reflecting business reality (of activities, plans)	Rapid state change	Inventory, status of unfinished parts
Potential	State reflecting business reality (of resources)	Moderate rate of state change	Process capability of equipment, equipment availability

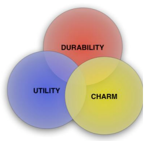
DEPENDENCY ▼

Knowledge Level

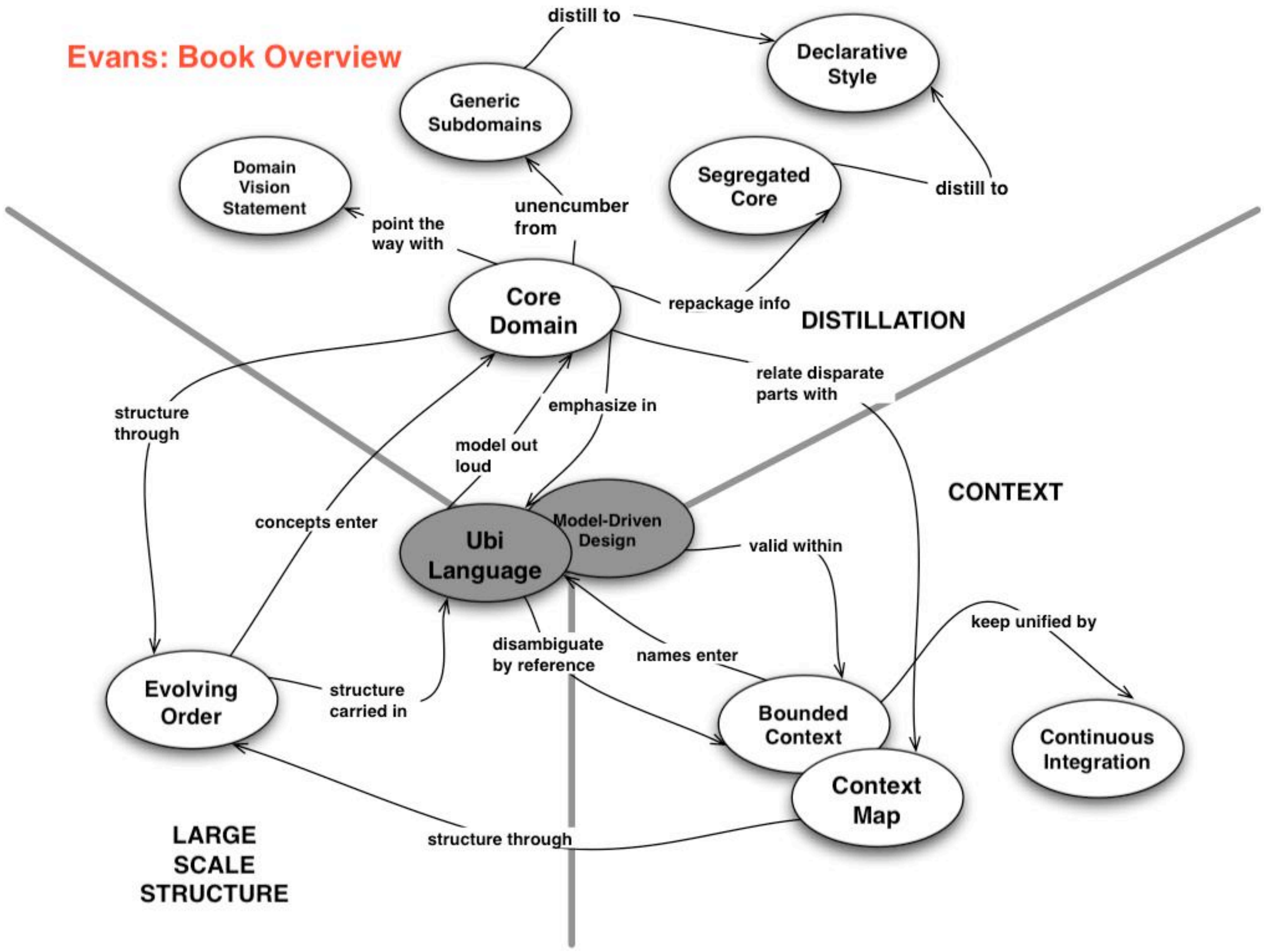
- “Meta Knowledge” A group of objects stating how another group of objects should behave
- Requirements for software with configurable behavior
- Another model about the model -- reflection, making software self aware
  - Base level that carries operational responsibility for the application and a meta level representing knowledge of the structure and behavior of software (super user customization, controlling operation of base level)

Clarification

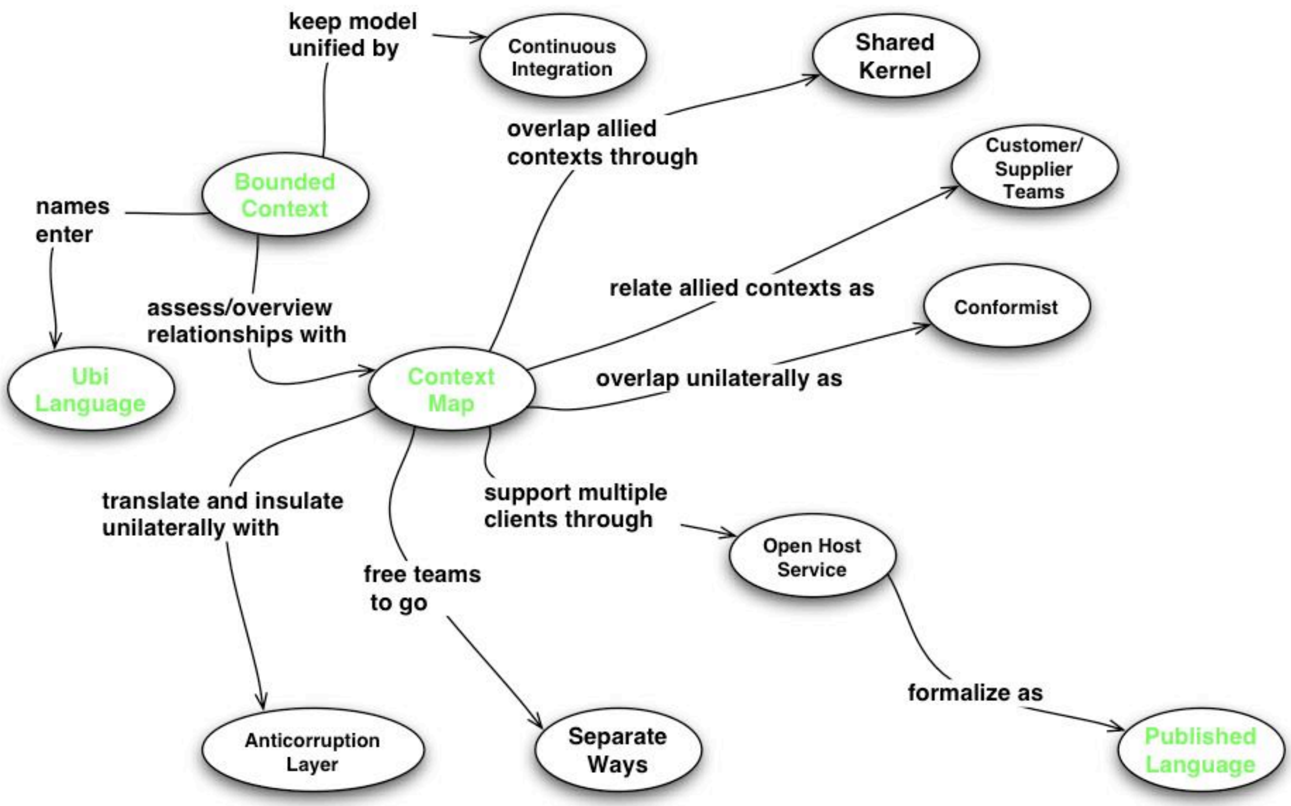
- Model Integrity vs Distillation vs Large Scale Structure
  - Model Integrity: UBI Language + Bounded Context + Context Map (Shared Kernel, Customer/Supplier Relationship, Conformist, Separate Ways, AntiCorruption Layer) + Published Language (affects development)
  - Distillation -> Abstract Core, Declarative Style (affects development)
  - Large Scale Structure -> System Metaphor + Responsibility Layers (affects understanding/decisions, project management, buy build decisions)
- Conceptual Contours vs. Bounded Contexts



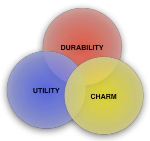
Evans: Book Overview



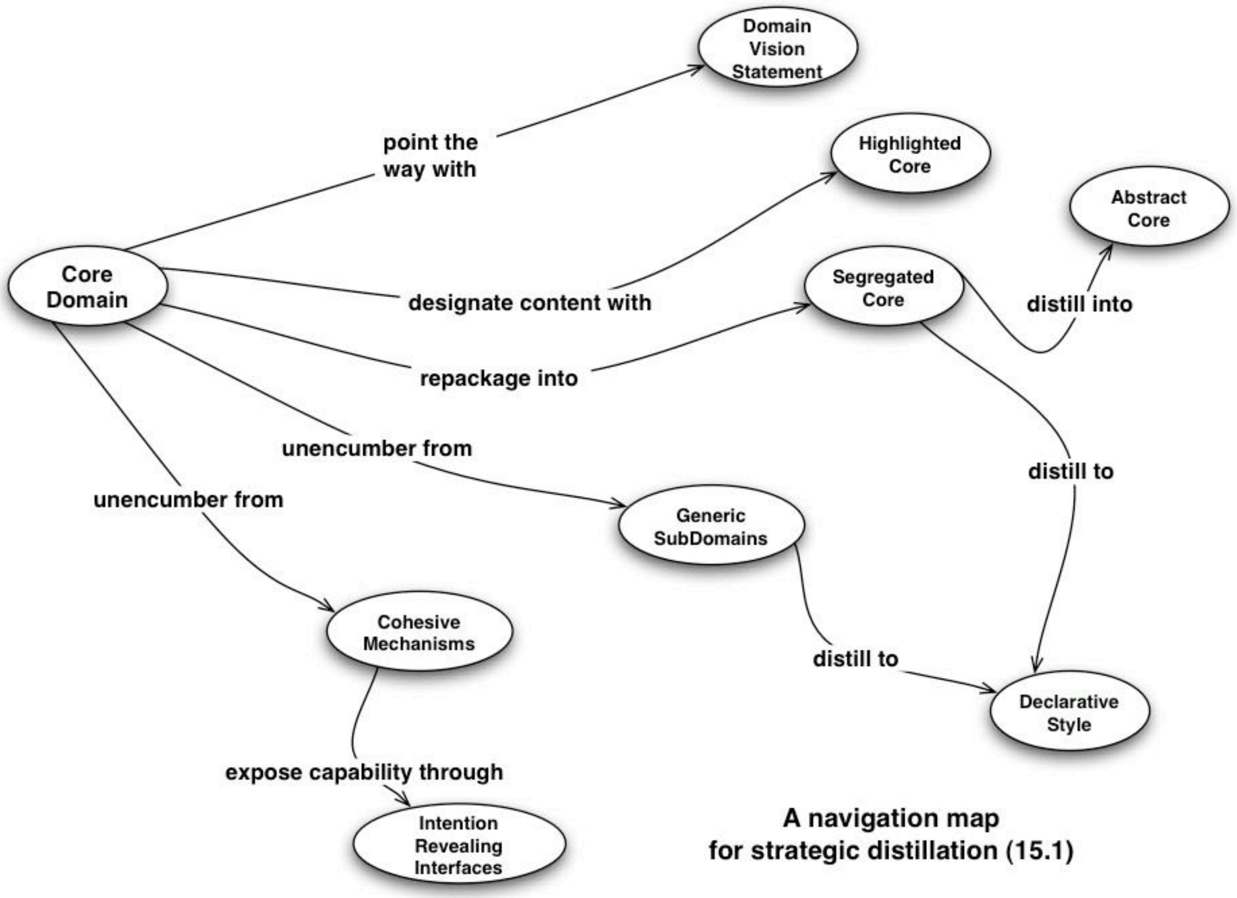
Model Integrity



A navigation map for model integrity patterns (14.1)



Distillation

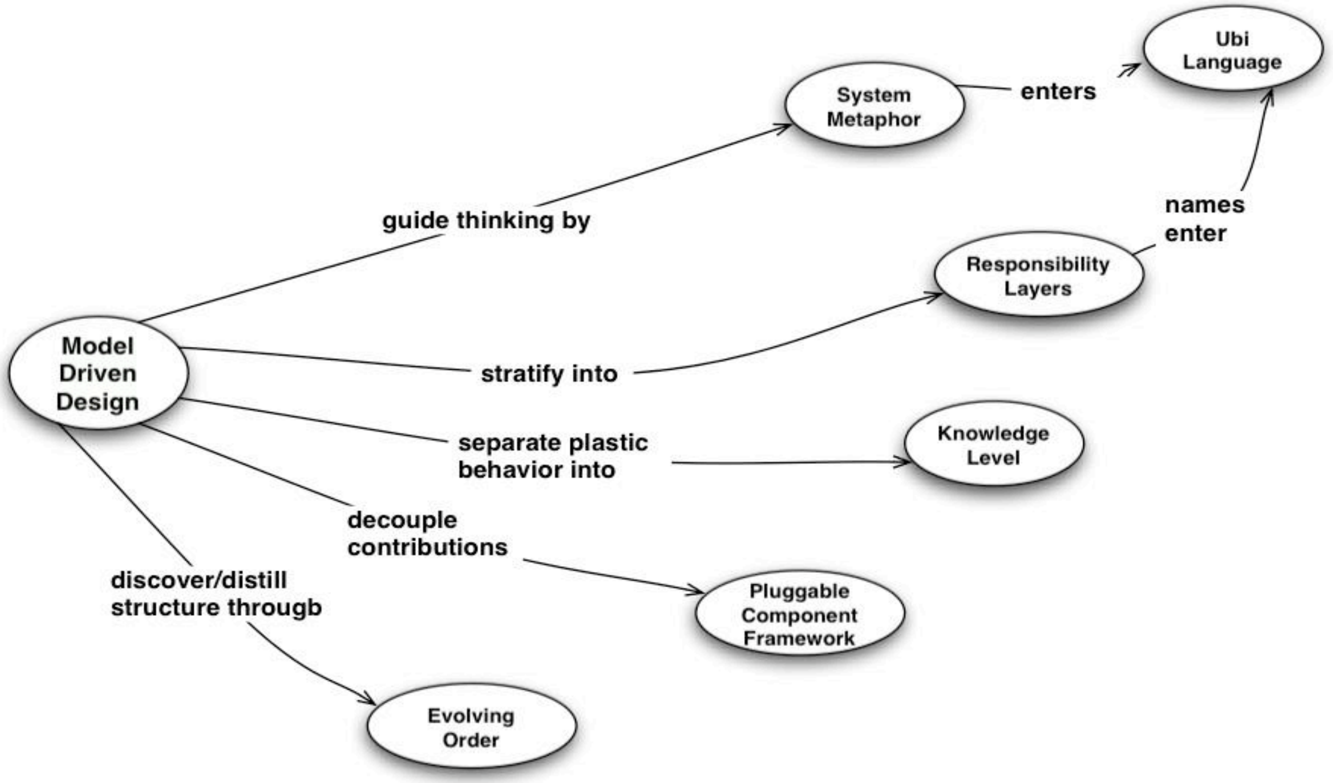


A navigation map
for strategic distillation (15.1)

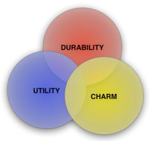
Large Scale Structure

- Lets you discuss system in broad strokes
- Avoids forest- trees syndrome, inhomogeneity of technical understanding syndrome
- A pattern of rules or roles and relationships that span the entire system and allow some understanding of each part's place in the whole -- without detailed knowledge of the parts responsibility
  - Understand the relationships in the system without understanding the details

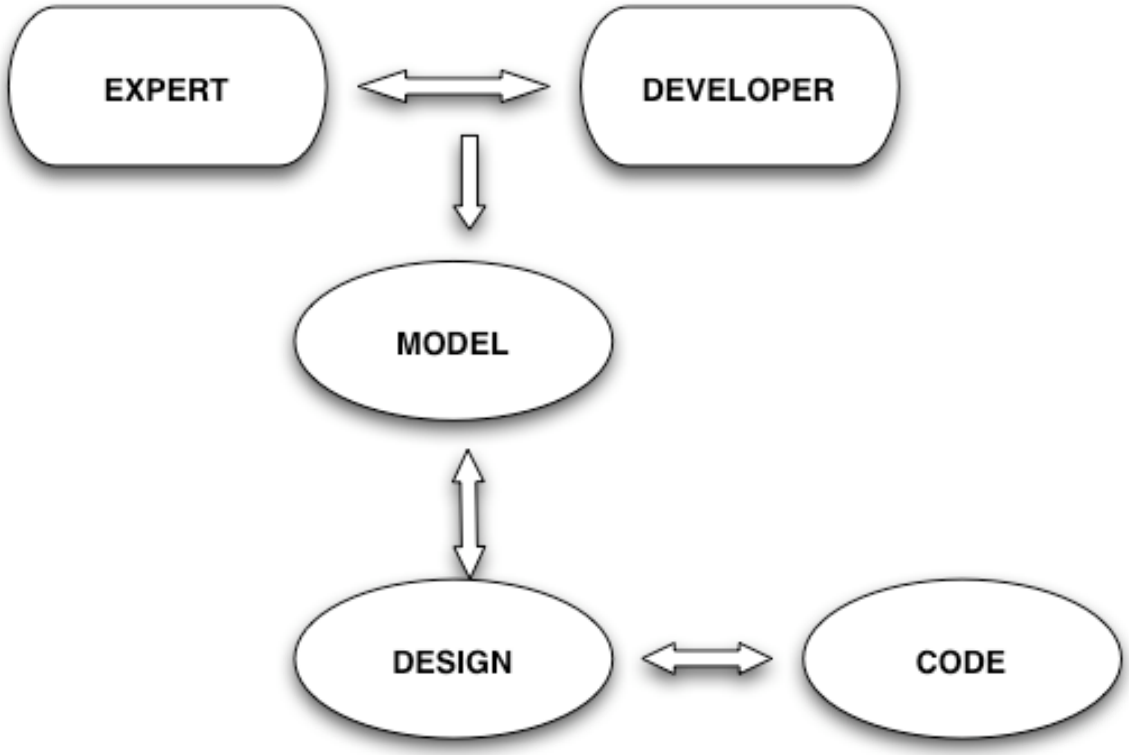
Large Scale Structure



Large Scale Structure (16.1)


DURABILITY
UTILITY
CHARM

30,000 Feet

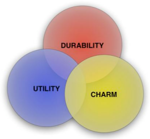

EXPERT
DEVELOPER
MODEL
DESIGN
CODE

DURABILITY

UTILITY

CHARM

Evans



Other References

- My draft paper on downstream systems!
- Maeda, J. The Laws of Simplicity, MIT Press, 2006.