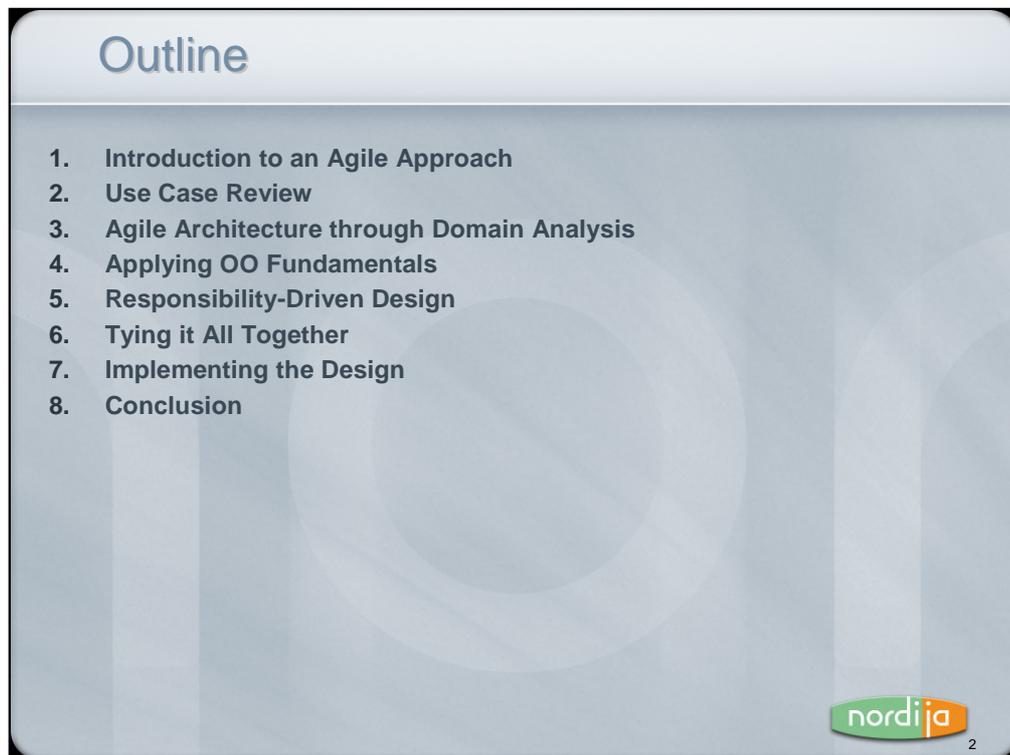


Copyright ©2006, 2007 Nordija A/S. All rights reserved.

3 hours. Target: 60 slides, 3 minutes/slide

Agile development usually talks in general about work queues and waves its hands about architectures all while singing the praise of user stories. How is a C++ programmer to reduce these to practice? This tutorial describes how to combine three techniques to achieve business objectives that go beyond the Agile tradition. The techniques are well known -- Use Cases, Domain Analysis, and CRC cards -- but they are combined in interesting and powerful ways to create a flexible, lightweight design method. The method helps projects create long-term business assets through lightweight front-loading of design. It reconciles user requirements with the architecture by separating role design from class design. The technique has been proven in many long-standing development projects ranging in size from a handful of people to teams comprising multiple departments. While the tutorial targets a C++ audience, the approach generalizes to Java development as well.



This outline presents a framework for a course on object-oriented analysis and design. This is intended to be a practical seminar that prepares people to “hit the ground running” after they are done. The course materials build on a foundation of concepts that support a level of understanding that will discourage students from applying the course ideas blindly.

The course is not tied to any specific methodology or CASE tool. There are many “tools” offered in this course for problem identification and definition (the Satir change model), for capturing user wants and needs (use cases), and for reconciling structural and functional needs (CRC cards). The enterprise may want to use its own tools, such as Rose, to present source code in graphical format for ongoing solution domain analysis. However, those tools are central neither to the design principles nor to the design processes at the core of effective analysis and object-oriented design, and they are given their proper place subordinate to the core course concepts.

The day boundaries are approximate; this is a custom packaging that has not previously been time trialed. The schedule and material are flexible. It is our goal to tune this as best as we can to your needs, so please give us your feedback on parts you see as weak or missing.

Prerequisites:

The student should be familiar with basic Java and C++ programming constructs including member functions, inheritance, and information hiding. The student should also be familiar with a typical software development life

Course Scope

- **Agile: A set of values**
- **Methods that precipitate from Agile**
 - XP
 - SCRUM
 - Crystal Clear
 -
- **These are largely project management methods**
- **Those are fine — but what are the tools that support *design*?**
 - Design and people are the keys to success
 - Methods are just there to support design
 - You also need design tools
- **This is a course about Agile design**
 - Agile analysis / design tools such as CRC cards and Use Cases
 - Adaptive/Agile implementation techniques including software investment and roles
- **The focus here is on developers**
 - Tools for use within a Scrum sprint, or to interface to the project management tools of Scrum and XP



3

People commonly interpret “Agile development” to mean something other than what we find in the Agile manifesto: that it is chaotic development, or that it is just development without documentation, or that it is XP. In fact Agile development is a set of values. Most articulations of Agile development focus on project management. Here, we focus on the developer: the tools they use to interface with project management constructs, or to support the work within an Episode or Sprint. Each of these tools derives from the values of Agile development. This course will highlight Agile values and practices with an Agile clown overlay as we go through the slides.

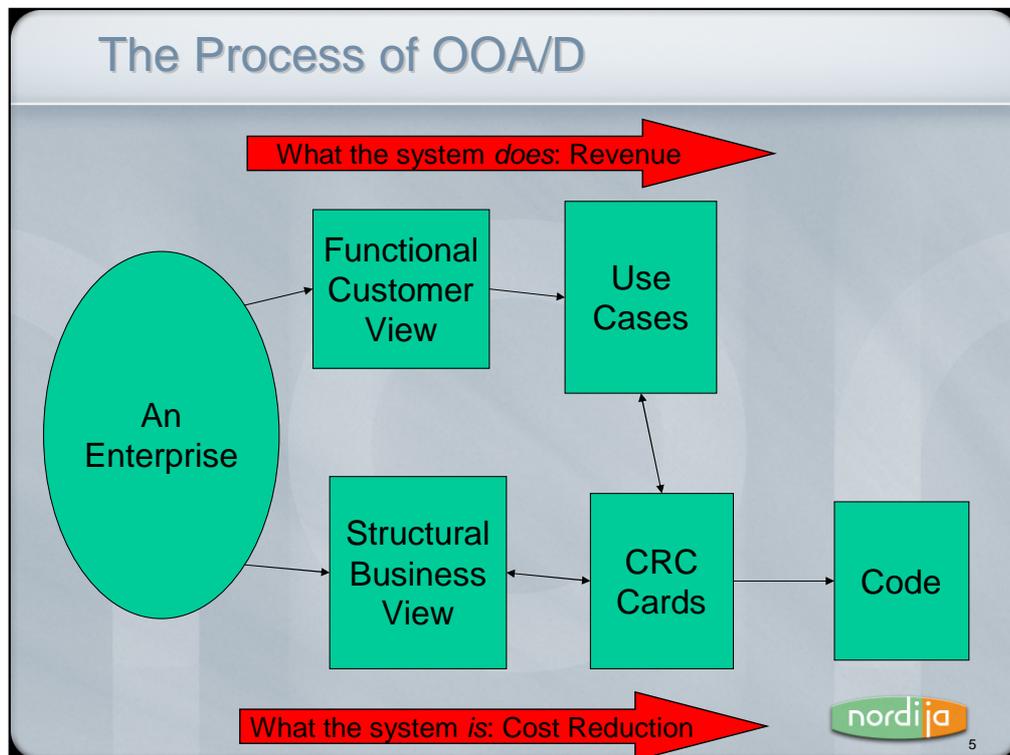
1. Introduction to an Agile approach

- **Processes and tools...**
 - ... replaced by teams that form around and nurture areas of specialization and deliverable features
- **Comprehensive documentation...**
 - ... replace by lightweight domain models (which is better than no documentation), a testable architecture, and an “executable specification” in CRC cards
- **Contractual agreements...**
 - ... replaced by direct interaction with domain experts, end users who bring Use Cases, and other stakeholders
- **A planned process...**
 - ... replaced by an environment that responds to changes in end-user needs with feedback that can go all the way into the architecture, and which is explicitly designed to support a broad spectrum of clients



4

Agile supports a system thinking approach. It replaces processes and tools by teams that form around and nurture areas of specialization and deliverable features. We want to build on that specialization. We replace documentation by lightweight document models, a testable domain model, and an “executable specification” in CRC cards, that moves testing from the technical domain to the social domain. We replace contracts with dialogue—both with domain experts during the creation of an Agile architecture, and through Use Cases that map out customer needs and wants. And instead of a planned process we prefer an environment that provides powerful and fast paths for feedback to propagate where it needs to go.



Here is a development model based on the need to increase revenues and decrease costs. The functional customer view, using Use Case models, is about capturing end user and customer wants and needs. If we do that and if we can satisfy those needs, we can increase revenues.

The structural business view is about building a robust architecture that is resilient in light of change. It is about making investments in the structure of our system that will reduce cost and have payoff in the long term.

Different structures arise from each of these. One architecture helps support profitability. Another reduces cost. Historically, the object paradigm has waved its hands and ensured us that objects achieve both these ends. That is just naïve. And it's a bit frightening: How do we achieve such a level of integration? Experience has shown that CRC cards provide a good foundation for solving this problem. But we need a little bit more to express the cross-cutting of these two business views in the vulgar code of the implementation. That's where *roles* come in.

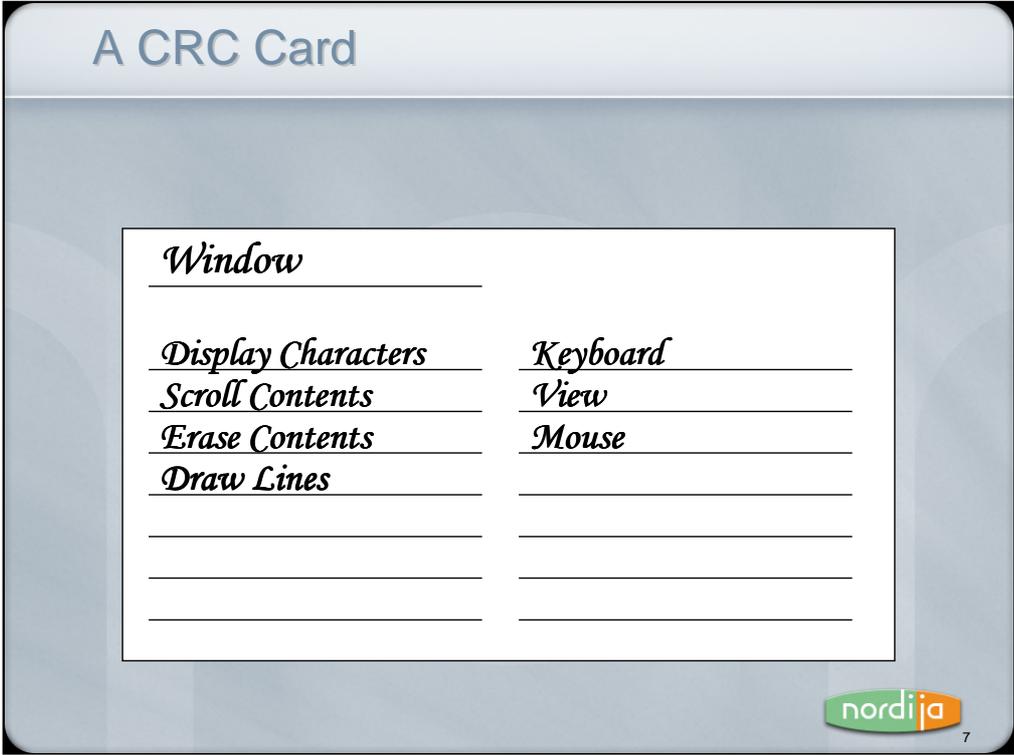
A Use Case

Name: <i>Get Paid for my Car Accident</i>	Goal: <i>Get paid for car accident</i>
Scope: <i>Business</i>	
Level: <i>User</i>	
Preconditions: <i>Policyholder has filed claim</i>	
Success Condition: <i>Insurance company pays claimant</i>	
Failure Condition: <i>Insurance company does not pay claimant</i>	
Trigger: <i>Claimant calls for the fifth time about a claim</i>	
Notes	
<i>Variations: Claimant may be a person or another insurance company or agency; payment may be by check or interbank transfer</i>	
Scenarios	
1. <i>Claimant submits claim with substantiating data</i>	
2. <i>Insurance company verifies claimant owns a valid policy</i>	
3. <i>Insurance company assigns agent to handle the case</i>	
4. <i>Agent verifies all details are within policy guidelines</i>	
5. <i>Insurance company pays claimant</i>	

From <http://members.aol.com/acockburn/papers/usecases.htm>



You know Use Cases from your earlier training. We will offer a short review of Use Cases in this seminar. Here is a sample use case following a form offered by Alistair Cockburn. Use Cases are informal analysis tools, useful for making models of what is. What we model is the user conceptual model of the workflow. We can make such a model with respect to a system that a user is already using and which we will augment or replace with new software, or we can make a general model of the user environment.



Here is a CRC card. You probably know CRC cards informally from your past training. Here, we will look at the CRC card technique in depth. First, you will learn refined facilitation techniques for CRC card sessions. Second, you will learn how to use them to model roles. Role modeling fits strongly in the original Use Case tradition. Roles will be crucial to our design process, and to the problem of fitting together business needs with customer needs.

2. Use Case Review

- **A collection of possible scenarios between the system under discussion and external actors, characterized by the goal the primary actor has toward the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail.**
- **A *contract* for the behavior of the system under discussion**
- **Why use cases?**
 - They provide a framework for **making difficult decisions early**
 - Prune the decision tree early
 - Making difficult decisions late:
 - Reduces your options, so your decisions are easier to make
 - Greatly reduces the quality of the decisions.



Here, we review Use Cases. Here are two definitions of Use Case. Unlike user stories, Use Cases are a contract. Use Cases have structure that goes deeper into system concerns than user stories do. Alistair Cockburn says, “The authors of Extreme Programming (XP) stayed with the idea of informal scenarios not having any formal structure at all. Kent Beck created the term user story to describe these sorts of requirements. A user story consists of just a phrase or a few sentences written on an index card, announcing something the user wants to do. In XP, the user story is not used as a requirements specification, but as a marker for a future conversation. Therefore, the card only needs to record enough information so the programmers and customer know what to discuss later.” --

http://alistair.cockburn.us/index.php/Use_cases,_ten_years_later

A Simple Use Case

Name: <i>Get Paid for my Car Accident</i>	Goal: <i>Get paid for car accident</i>
Scope: <i>Business</i>	
Level: <i>User</i>	
Preconditions: <i>Policyholder has filed claim</i>	
Success Condition: <i>Insurance company pays claimant</i>	
Failure Condition: <i>Insurance company does not pay claimant</i>	
Trigger: <i>Claimant calls for the fifth time about a claim</i>	
Notes	
<i>Variations: Claimant may be a person or another insurance company or agency; payment may be by check or interbank transfer</i>	
Scenarios	
1. <i>Claimant submits claim with substantiating data</i>	
2. <i>Insurance company verifies claimant owns a valid policy</i>	
3. <i>Insurance company assigns agent to handle the case</i>	
4. <i>Agent verifies all details are within policy guidelines</i>	
5. <i>Insurance company pays claimant</i>	

From <http://members.aol.com/acockburn/papers/usecases.htm>



This is my preferred format for Use Cases. It can fit on a large index card, and balances richness of description with simplicity.

You may prefer to keep these on-line with a tool such as Plan B from Nordija. However, it is good to print out individual cards during meetings; there is something powerful in the meeting dynamics that arises from the tactile nature of a card.

Use Cases in Development

- **Support responsibility-driven design**
 - Responsibilities are explicit
 - Scenarios exercise the responsibilities
- **Before deployment, prepare testing**
 - Re-identify primary actors
 - Primary actors “run” their Use Cases
 - ...or let use cases drive automated testing
- **Monitor project status by use case goals**
 - Something missing from User Stories
- **Attach non-functional requirements to goals**
- **Get subtle requirements from goal failures**



10

Why do we use Use Cases? Use-cases are responsibility-based; we can glean responsibilities from them to drive responsibility-driven design. Responsibility-driven design leads to maintainable designs and is long recognized by Wirfs-Brock, Beck and others as leading to good object-oriented development. Having Use Cases, you can simulate the design in your head even before writing code, ensuring that all the necessary interfaces are present. We will discuss CRC cards as a tool for responsibility-driven design tomorrow.

3. Agile Architecture through Domain Analysis

- **Creating Architectural Foundations**
- **Building software families instead of one-off systems**
- **An investment for the future**
- **Usually not incremental or Agile, but can be made lightweight**
- **Lightweight — at most one sprint of work (about a week)**



Did you know that XP condones the creation of an up-front architecture? See *Extreme Programming Explained*, p. 113. Using domain knowledge is better than XP's idea of using "speculation"



11

Now we've covered the "what the system does" part of design, let's turn our attention to "what the system is." This is about creating an architectural foundation that is a long-term investment that underlies and enables user feature delivery.

The main value of domain analysis is in supporting multiple customers, either at once (a bad idea for starting) or over time (usually inevitable). We aim to build families of products: to build a product line instead of one turnkey product.

You usually do implicit domain analysis at the beginning of every project, either by talking with clients or domain experts, or relying on your own knowledge. We add a bit of disciplined thinking to that process. You can make it as extensive or as efficient as you like. But we strongly advise generating a bit of lightweight documentation of your domain structure at the beginning. The main value of this documentation is to defocus you from the scenario at hand and to focus on the system structure: systems thinking. The secondary benefit is to create mementos as a record of architectural decisions you may choose to later revisit.

A key concept: Scope

- **Relates to business identity**
- You should *analyze* to the boundary of your business
- You should *articulate analysis* to the boundary of your business
- You should *design and implement* to the boundary of paying customers (or speculative development)
- **The bad news:**
 - It's difficult to get three customers together in a room to discuss shared ideas
- **The good news:**
 - Experienced architects and developers often know the scope
- **Scope is ultimately a business decision**
- **Use Case extensions delineate the project scope**



nordija

12

You can't be all things to all people, so you need to scope your analysis. This is not a technical question, but a business question. You can't be every business in the world: analyze to the boundaries of your business.

Your architecture will capture the structure of your domain analysis. As you flesh out the system, the class interfaces will start to have filled-out member functions. Some of these you can do at the beginning from your domain knowledge alone, pulled by some foreknowledge (provided by the Use Cases) that that code will be needed. But you can declare an interface without implementing it. You should design and implement only to the boundary of the Use Cases—NOT that of the domain.

How much architecture?

- **Why do architecture?**
 - The customer doesn't care
 - The system doesn't care
 - Architecture supports the organization, the GUI, and a few implementation constraints (e.g., physical distribution)
- **How long?**
 - In six months, you can nail it
 - However, that goes into many details that, though part of long-term common code, does not affect the organizational structure
- **Therefore:**
 - Do just enough up-front architecture to:
 - Stabilize the organizational structure
 - Drive the human/computer interface and
 - Map implementation constraints



13

Emerging Success of DA Techniques



SUNGARD CSA BLOG

Darren Wesemann

Commonality and Variation, key CSA principles

March 23, 2006, 9:43 am

Posted by Darren in Software

Rating: 0/5 Votes : 0

The principles of commonality and variation are well defined in a thesis paper by James Coplien "Multi-Paradigm Design", (can be downloaded from: <http://prog.vub.ac.be/Publications/2000/vub-prog-phd-00-01.pdf> also described in his book: "Multi-Paradigm Design for C++", published by Addison-Wesley in October of 1998 [Coplien1999], the thesis paper is essentially the book). These principles are fundamental to SunGard's success with the CSA, and are relevant to software architecture as well as product management going forward.

As I've discussed in earlier blogs, the CSA uses abstraction heavily in order to make sense of the various elements and concepts found in our myriad of architectures at SunGard. Coplien's paper addresses abstraction techniques in terms of commonality and variation since complexity and problem definition are key issues in software work today. For example, grouping is a useful technique in abstracting similar concepts. Procedures or methods form out of grouping steps of an algorithm by their relationships, and responsibilities of an encapsulated collection of related data are grouped to form classes.

Coplien's design approach uses analyses of both the application and solution domains in parallel. It provides techniques in finding solution domain constructs that most naturally express the structure of the application domain. It seeks a match between commonalities and variabilities of the solution domain with those of the application

Sep 2006
S M T W T F S
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
<< >>

Recent posts

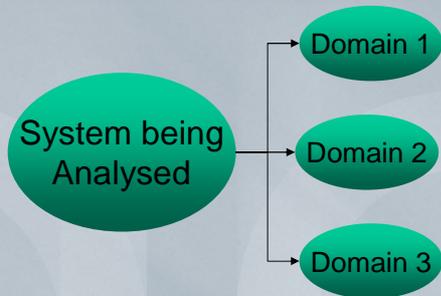
Collaborative SOA
Lessons Learned
Software-as-a-Service vs.
ASP
Is Eclipse finally getting
cozy with Sun?
Iona's open source ESB
All Buses Are Not Created
Equal
Where'd the
Hub-and-Spoke model
go?
Commonality and
Variation, key CSA
principles
CSA integration and
preserving legacy
Reuse, not so illusive
anymore, thanks to
process
SOA ROIs



14

Domain analysis techniques are enjoying resurgent success (why? See my my musings at <http://www.artima.com/weblogs/viewpost.jsp?thread=167119>.) Here is an example that uses the techniques that underlie this course.

All Design starts by Divide and Conquer



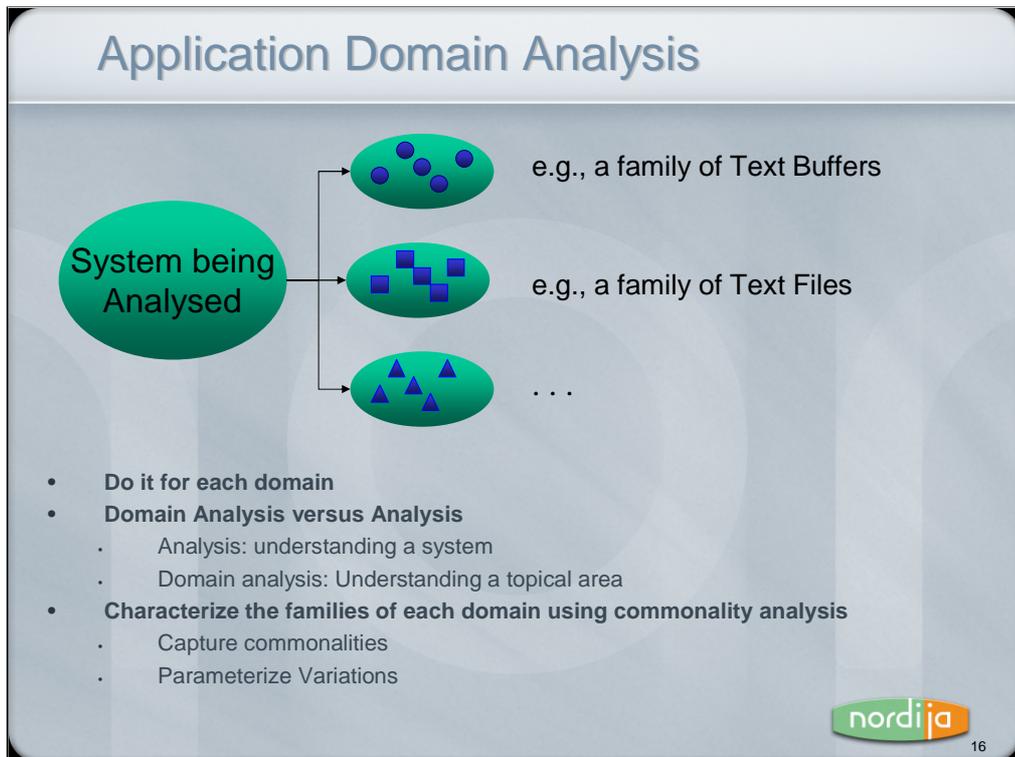
- **The units of division are *domains***
 - Not objects — that prejudices the implementation
 - Not modules — that is a design concern
- **Partitioning follows:**
 - Business intuition and history
 - The need for software families within the product
- **The partitioning can never be perfect—more about that later**
- **Involve all stakeholders from marketing to developers!**



15

One goal of analysis and design is to manage complexity. A good analysis attacks complexity by organizing information without losing information. The first step in organizing information is almost always divide-and-conquer. We too often divide the system using prejudiced implementation techniques: e.g., we divide the system into functions or objects or modules. Instead, we should let our intuition speak to us about the overall domain structure of the system.

Domains often (but not always) arise from families or product lines. For example, if we are building a text editor, we might build families of disk file types or of editing language types. Each one of these is a domain.



Once we divide our system, we look at each domain to discover its “shape.” We carefully observe the family members: why did we group them? We will have grouped some of them by their commonality of behavior and structure, as we might do with Text Buffers. We might group some of them because they share the same algorithm, or because they share the same structure. We also want to regularize the way that family members vary. For example, we can look at a family of sorting procedures that all use the same algorithm but which vary according to the type of the elements being sorted. The type of the sorted element is called a parameter of variation for the domain.

Example: Text Buffer for a Text Editor

- **All text buffers share common behavior:**
 - Yield line at line N
 - Initialize from some file or stream
 - Write to some file or stream
- **All text buffers share common structure:**
 - The top line in the buffer
 - The number of lines in the buffer
 - “Dirty” bit
- **Text buffers can vary in the following way:**
 - Character set (`wchar_t`, ASCII, EBCDIC)
 - Memory Management Algorithm

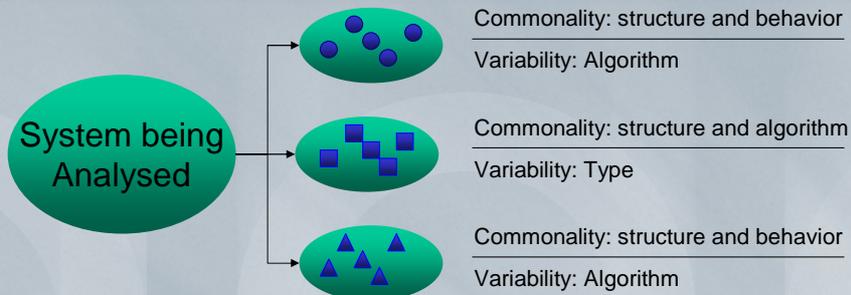


17

We can look at Text Buffers as an example domain. We might build many different text editors for different customers, and their requirements on Text Buffers will vary; yet all Text Buffers have something in common (which is why we grouped them as Text Buffers). All Text Buffers can be asked to yield a given source line, or to initialize themselves from some file, or to write themselves out to some file. They may all share common structural elements such as the line number of the top line currently in the buffer, or a flag indicating whether the buffer has been modified.

Text buffers vary in regular ways. Each Text Buffer supports a specific character set and a particular memory management algorithm. These two features are said to be parameters of variation.

Application Domain Analysis



- **Look for *kinds* of commonality:**
 - Structure
 - Name / meaning
 - Value / state
 - Algorithm
 - ... very few of these *kinds* of things
 - These go to the foundations of human perception
- **Do the same for variation**



Domain Dictionary



18

Once we have delineated the commonalities and variations, we classify them according to a taxonomy of design. This taxonomy is fairly universal across Western thinkers and forms the foundation of the features of all practical programming languages. There is a remarkably small number of such considerations: structure, name/meaning, value/state, algorithm, and type. These go to the foundations of Western cognition. We call them commonality categories.

For each domain, we characterize both the commonalities and parameters of variation according to this model.

Products of Domain Analysis

- **Domain Documentation**
 - Not UML
 - Use Cases aren't for domain knowledge, but for current customer desiderata
 - Objects alone prejudice understanding important domain configurations of commonality and variation
 - Instead, document:
 - **Software families**, and for each family:
 - A basic **domain vocabulary**: very simple
 - The **commonalities** across family members
 - **Parameters of variation** that distinguish family members
- **A baseline architecture**
 - Real code that can compile and demonstrate trivial functionality
 - Partially filled out
 - Do "Use Case YAGNI"



19

What do you have at the end of a domain analysis? Remember, this is “Agile architecture,” so we don’t want a ton of documents. But we do want domain documentation: it captures one of the most important assets of an enterprise, and in fact can be a major component of a knowledge management program.

We don’t want to use objects to do this: they prejudice the solution. We instead look for configurations of commonality and variation, for software families. We document these using commonality and variation.

As a result we will have a lightweight set of documents that describe each of our software families (maybe a half-dozen or so, one page each). We can take these ideas into C++ or Java class interface declarations that actually can compile against each other. If we stumble onto some generic knowledge about implementation along the way, we can deliver that as part of the architecture as well. That might include COTS software—the way having been pointed by domain analysis. Do just enough Use Cases so that the system integrates and initializes in a sound way, and can demonstrate trivial functionality.

This framework will have many pure virtual (deferred) functions that may not yet be overridden in derived classes. Avoid the temptation to implement them on the basis of some guess of how they will be involved in a future Use Case. Let the real Use Cases drive code generation.

An Example: Text Editing Buffers

- **Commonality: Behavior, some data structure**
 - All text buffers can be asked to yield a given line, to fill themselves from an input type, to write themselves to an output type
 - All text buffers know the line number of the top line that they have contain
 - All text buffers have a “dirty bit”
 - All text buffers contain a line count
- **Variabilities:**
 - Character Set (Type)
 - Working Set Management (Gross Algorithm)
- **Solution:**
 - Templates (Common data structure, different type)
 - Inheritance with Virtual Functions (Common behavior, different functions)



20

Let's return to our running example, text editing buffers. What did we find out about them during commonality analysis? that all family members share behaviors and some data structure. Against that background of commonality, individual text buffers exhibit regular variabilities: the character set they support, the working management algorithm they support, etc. We can select commonality categories for these parameters of variation: character sets are a type, and working set management is gross algorithm.

We look up the commonality/variability pairs (structure and behavior/type; structure and behavior/gross algorithm) in the C++ solution domain analysis table, noting instantiation properties (not applicable in this case) and binding times (we can afford to bind these at compile time). We find that templates and inheritance with virtual functions are indicated as the appropriate solution mechanisms.

Domain Dictionary for Text Editors

- **Editing Language:** the keystroke sequences and menus accessible to the user (vi, emacs...)
- **Text Editing Buffer or Text Buffer:** manages the text in between its residence in a file
- **File:** long-term storage for the text on a secondary storage device
- **Window:** medium for displaying the edited text to the user for interactive editing
- **Input Device:** keyboards, pointing devices, and other facilities for user input to the editor
- **Command:** a human/machine interface gesture



21

A domain dictionary is a simple dictionary of domain terms. Some of them may be domains, while others may be more detailed definitions useful to the designer. If in doubt about whether a term belongs in the domain dictionary, include it.

TextBuffer Variability Table

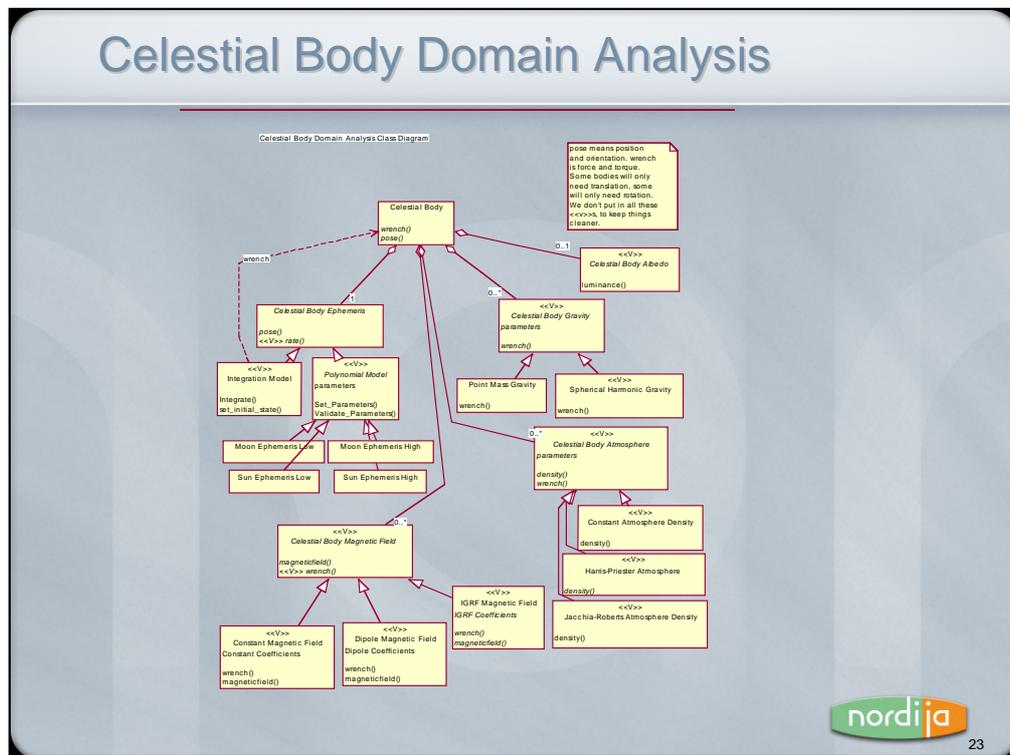
TextBuffer: Common Structure and Algorithm

Parameters of Variation	Meaning	Domain	Binding	Default / Technique
Output Type	The formatting of text lines is sensitive to the output medium	Database, RCS, TTY, UNIX file	Run	UNIX File
Character Set	Different buffer types support different character sets	ASCII, EBCDIC, wchar_t	Compile	ASCII
Working Set Management	Different applications need to cache different amounts of memory	Whole file, whole page, LRU, fixed	Compile	Whole file
Debugging Code	Debug in-house only, but keep tests in source code	Debug, production	Compile	None

22

This table captures the parameters of variation for Text Buffers, their meaning, the domain of variation, the binding time, and the technique that we draw from the table of C++ commonalities and variabilities. This table documents our design decisions and points the way to implementation structures that support the “shape” of the analysis architecture.

Celestial Body Domain Analysis



From McComas et al, "Addressing Variability in a Guidance, Navigation, and Control Flight Software Product Line," citeseer.ist.psu.edu/418569.html

Changing the analysis into architecture

- **Programming languages express Commonalities and Variations**
 - Object orientation
 - Commonality in behavior and structure
 - Variation in algorithm and structure
 - Procedural
 - Commonality in algorithm
 - Variation in parameters
 - Templates
 - Commonality in code structure
 - Type and value parameters
- **Other technologies also can express families**
 - Parser generators, GUI generators, etc.
- **We need to take the commonality analysis into an architecture**
- **We do that in a process called transformational analysis**



24

Once we are done capturing commonalities and variations, now what? It's time to code it up.

We know the commonalities and variations of each of our domains. Our architecture should express those. Why? Because what is written is hard to change. And because the commonalities of our domain, which should be long-term stable invariants, won't change. And the ways in which the family members vary isn't likely to change.

How do we express these? In our implementation technologies. The most common implementation technology is a programming language. There are other implementation technologies, created to capture the structure of specific domains: parser-generators, spread sheets, document processors, and the like. You need to know what your quiver of implementation structures looks like, too. Knowing that, you can translate your domain analysis into something that looks like code: an architecture. To do that, we try to match up the solution domain structures with the analysis structures.

Solution Domain Analysis

Commonality	Variability	Binding	Instantiation	C++ Feature
Function Name and Semantics	Anything other than algorithm structure	Source	N/a	Template
	Fine algorithm	Compile	N/a	#ifdef
	Fine or gross algorithm	Compile	N/a	Overloading
Data Structure	Value of State	Run Time	Yes	Struct, simple types
	A small set of values	Run time	Yes	Enum
	Types, values and state	Source	Yes	Template
Related Operations and Some Structure	Value of State	Source	No	Module
	Value of State	Source	Yes	struct, class
	Data Structure and State	Compile	Optional	Inheritance
	Algorithm, Data Structure and State	Compile	Optional	Inheritance
		Run	Optional	Virtual Functions

- Think of this as *meta-design*
- Gives a formal sense of what "paradigm" means
- GOF patterns and special language features come from additional tables



25

Bjarne Stroustrup has never called C++ an object-oriented programming language.

We analyze not only the application domain, but the solution domain as well. This is the domain analysis for C++ commonality and variability. It is the same form of table we will use when seeking commonalities and variabilities in the application domain.

The table builds on commonality categories. When we find a particular commonality category and variability category in the application domain, we can look up that pair in this table and choose the corresponding implementation technique for the implementation technology being used. This table applies to C++ users; we can build other tables for other programming languages. I believe the tables work best for highly expressive languages; for example (but not exclusively) those with a strong type system. We can also build tables for implementation technologies other than programming languages, like finite-state machines, databases, etc.

TextBuffer Transformational Analysis

TextBuffer: Common Structure and Behavior

Parameters of Variability	Meaning	Domain	Binding	Default / Technique
Output Type Structure, Algorithm	The formatting of text lines is sensitive to the output medium	Database, RCS, TTY, UNIX file	Run	UNIX File <i>Virtual Functions</i>
Character Set Non-structural	Different buffer types support different character sets	ASCII, EBCDIC, FIELDATA	Compile	ASCII <i>Templates</i>
Working Set Management Algorithm	Different applications need to cache different amounts of memory	Whole file, whole page, LRU fixed	Compile	Whole file <i>Inheritance</i>
Debugging Code Code Fragments	Debug in-house only, but keep tests in source code	Debug, production	Compile	None <i>#ifdef</i> (from <i>Negative variability Table</i>)

We annotate the first column of the table with the commonality categories for the respective parameters of variability. Looking at the background of commonality (Common Structure and Behavior) and the commonality categories in the Parameters of Variability column, we can choose a C++ feature to express the commonality/variability pair by looking up the pair in the Transformational Analysis Table and Negative Variability Table.

In the TextBuffer design, we see that a combination of virtual functions, templates, inheritance, and `#ifdef` is called for.

The Solution for Text Buffers

```
template <class CharSet> struct TextBuffer {
    virtual Line line(const LineNumber&) const;
    virtual void write(File&);
    . . . .
private:
    virtual void pageManagement(const LineNumber&);
};

class EmacsBuffer1: public TextBuffer<wchar_t> {
    void pageManagement(const LineNumber &);
    . . . .
};

class EmacsBufferJap: public TextBuffer<Katakana> {
    void pageManagement(const LineNumber &);
};
```



27

Here is one solution for the text buffer design that exhibits how to capture the commonalities and variabilities in C++ language constructs. The conversion from the transformational analysis to code is not formal, but should be intuitive to the informed C++ designer.

Negative Variability Table

Kind of Commonality	Kind of Variability	C++ Feature for Positive Variability	C++ Feature for Corresponding Negative Variability
Name and Behavior	Gross Structure or algorithm (parametric)	Templates	Template Specialization
Structure, algorithm, name, behavior	Fine structure, value or type	Templates	Template argument defaulting
Enclosing data structure	Fine structure and "type"	Inheritance	union
Semantics and Name (of function)	Default value in a formula or algorithm	Argument defaulting	Supply explicit parameter
Commonality in some data structure, perhaps in algorithm	Membership in Data Structure	Overloading	Overloading
Some commonality in structure and algorithm	Behavior	Inheritance, adding data members	Re-factor using pointers to alternative implementations
Most source code	Fine algorithm	Inheritance, overriding or adding virtual functions	Private Inheritance
		#ifdef	#ifndef

Supplemental Material



28

For negative variability, we use this transformational analysis table instead of the one presented earlier for positive variability.

Most GOF patterns are indicated

Commonality	Variability	Binding	Instantiation	Pattern
Function name and semantics	Fine algorithm	Run time	N/A	Template Method
	Algorithm	Run time with compile-time default	N/A	Unification + Template Method
	Algorithm: Parameter of variation is some state	Run time	Yes	State
Related operations and some structure	Gross algorithm	Run time	N/A	Strategy
	Value of state	Source time	Once	Singleton
	Gross Algorithm	Source time (or compile time)	N/A	Strategy (templates) or Unification
Related operations but <i>not</i> structure	Incompatible data structure	Any	Yes	Bridge or Envelope/Letter

Supplemental Material



29

Most design patterns from the GOF book are stylized mechanisms to capture microarchitectures that represent particular commonalities and variabilities. Others are techniques to capture negative variability.

Patterns of Negative Variability

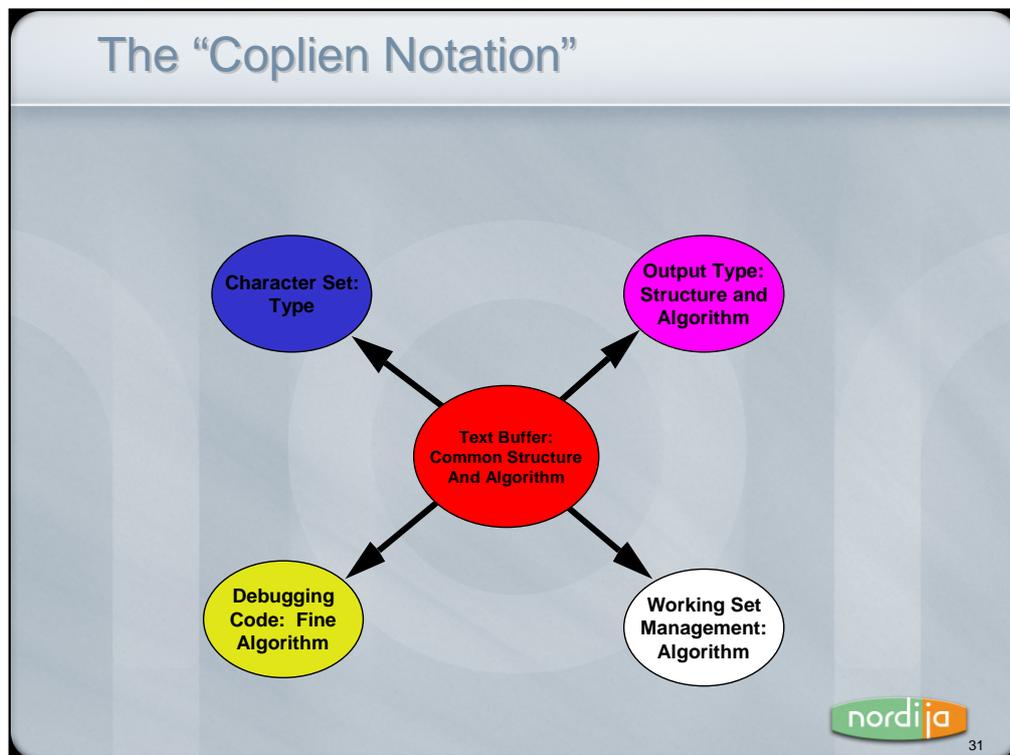
Kind of Commonalty	Kind of Variability	Binding	Instantation	Pattern
Some structure and algorithm	Function name and semantics	Compile or run time	Optional	Adapter
Related operations but <i>not</i> structure	Cancellation of class membership	Any	Yes	Bridge

Supplemental Material

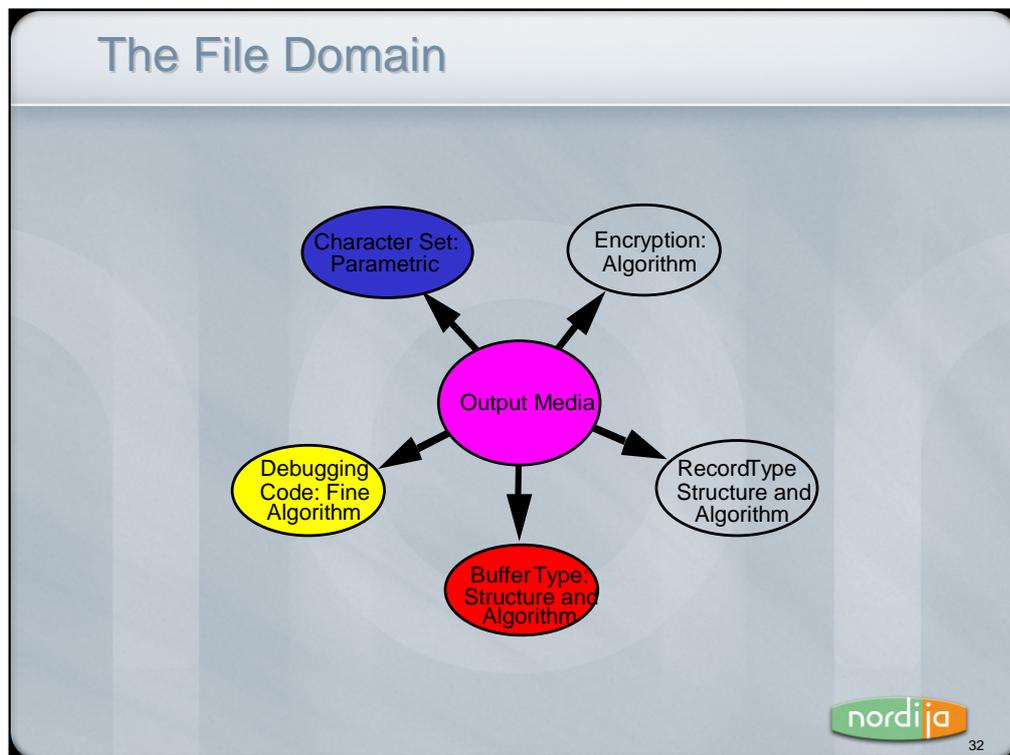


30

Many patterns capture commonly recurring configurations of negative variability.

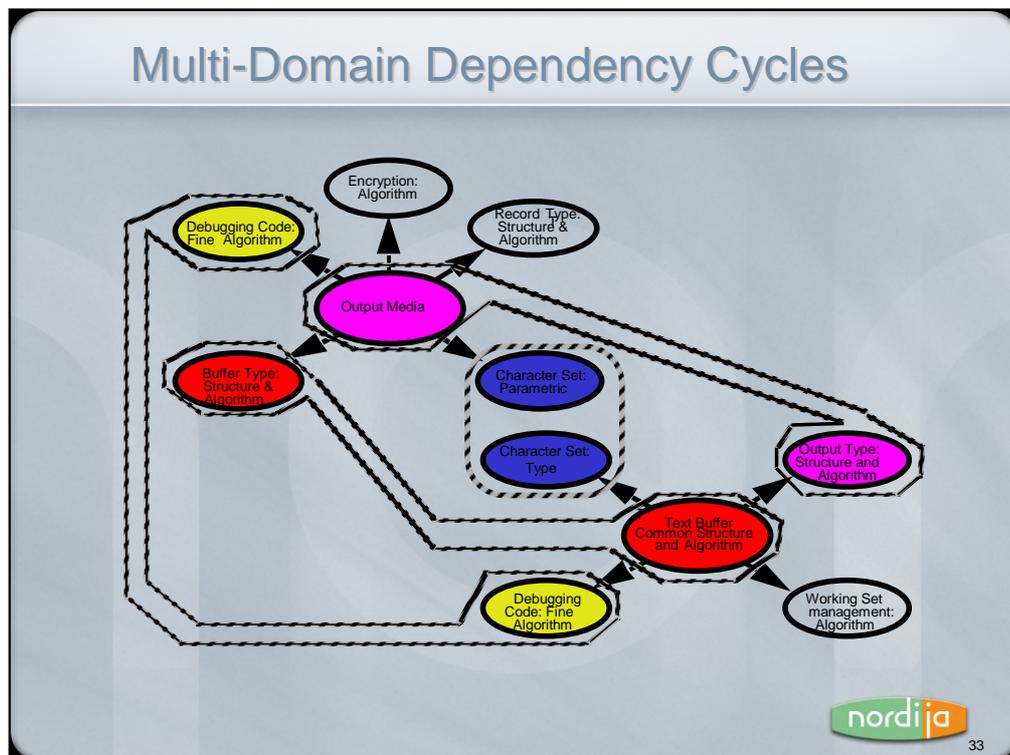


Here we encode the parameters of variation in graphical form. The commonality domain (Text Buffer) is at the center; we draw arrows to circles representing the parameters of variation. Why? This may look like a hobby horse that is done for its own sake, but we this notation will serve us well later. If you think about it, parameters of variation—which we describe as commonality categories—may be interesting domains in their own right. In a complex system, what is a parameter of variation in one domain may be the core commonality in another. That implies an interaction between domains. The best domains minimize these interactions, as we emphasized earlier in this seminar; however, some interaction is inevitable. These interactions can introduce strange loops into design, and it pays to understand them.

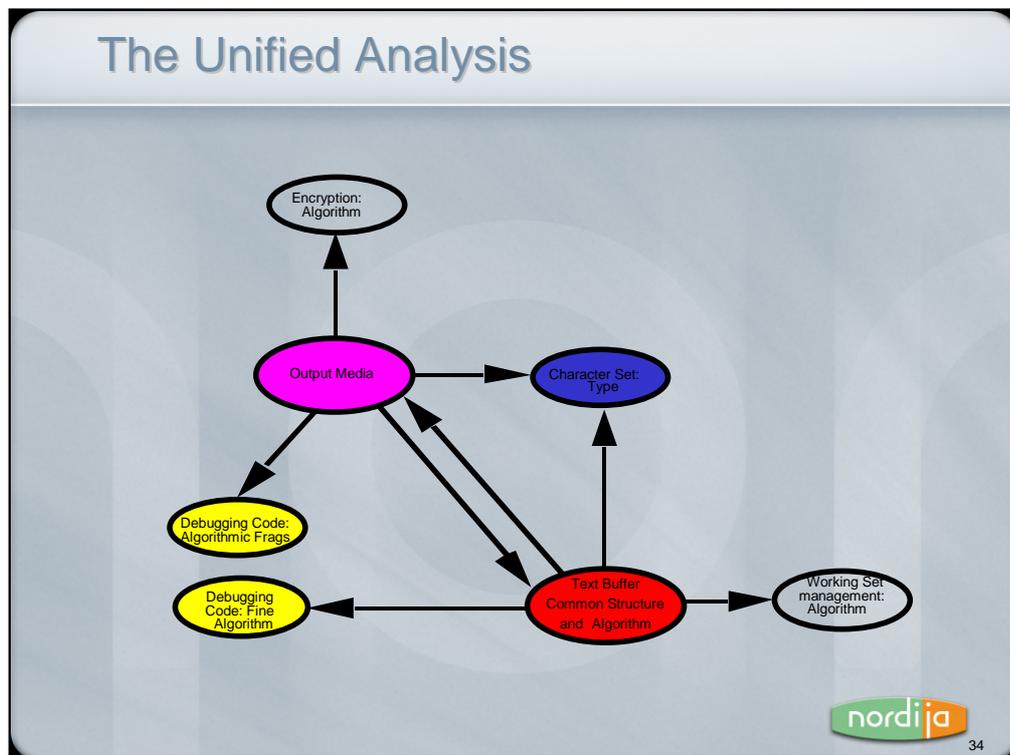


Here is a dependency chart for the **File** or **Output Media** domain. The parameters of variation include the Buffer Type—our Text Buffer domain. Remember that the Text Buffer domain also took **OutputMedia** as a parameter of variation. Each domain depends on the other! Furthermore, both domains take Character Set as a parameter of variation.

We should merge these graphs to understand the overall design of the text editor



To design an editor, we must reconcile the design of all relevant domains. Note that there is profound reuse implication here: the granularity of effective reusability depends on the ability to break domain dependency cycles. If we can't easily break these dependencies, then we must use both domains together. As these domains pull in more and more related domains, they become a large, reusable architecture. We sometimes call such architectures *frameworks*. Of course, the individual sub-domains may be frameworks, too, but they are only reusable if they are large enough to be interesting, and small enough to be manageable.



When we collapse the graph, the circular dependency between Output Media and Text Buffer is apparent. Other parameters of variation, like Character Set type, factor out nicely.

We can deal with circular dependencies either by 1) adding levels of indirection; 2) using unconventional abstracting techniques for one of the reciprocal arrows (delegation instead of inheritance; templates instead of inheritance), or 3) re-factoring the circularity out of the design.

Let's look at the implementation of this design, for the case where the Buffer Type depends on Output Type at run time, and the Output Type depends on Buffer Type at compile time.

One Potential Solution

```
template <class TextBuffer, class CharSet>
class OutputMedium {
public:
    void write() {
        ....
        subClass->getBuffer(writeBuf);
    }
    OutputMedium(TextBuffer *sc): subClass(sc) {}
protected:
    TextBuffer *subClass;
    CharSet writeBuf[128];
};

template <class TextBuffer, class Crypt, class CharSet>
class UnixFile: public OutputMedium<TextBuffer, CharSet>,
protected Crypt {
public:
    UnixFile(std::basic_string fileName, TextBuffer *sc):
        OutputMedium<TextBuffer, CharSet>(sc) {}
    void read() {
        ....
        Crypt::decrypt(buffer);
    }
};
```



35

In our example, the text buffer behavior varies according to the output type, and must defer the decision of which behavior to use until run time. The output type depends on the buffer type, but can bind its decisions at compile time. We address this using a multiple dispatch idiom. The text buffer notifies its associated output medium that it wants to perform an operation, and the output medium reciprocates.

We note that the output type depends on the buffer type at compile time. The domain is Output Media; the parameter of variation is the buffer type (variation in type); the underlying commonality for Output Media is structure and algorithm. Multi-paradigm design indicates that we should use templates. We handle encryption with inheritance; more about that later.

Output media also vary according to the record type: whether a UNIX file, database, RCS file, etc. This variation shows up in structure and algorithm. Requirements dictate that we track the variability at run time, so we use inheritance and virtual functions. The virtual declaration appears in the base class above. The write member function will be discussed below.

TextBuffers with typedefs

```
template <class CharSet>
class TextBuffer {
public:
    string getLine() {
        string retval;
        ....
        return retval;
    }
    void getBuffer(CharSet *) {....}
    TextBuffer() {....}
};

typedef UnixFilePagedTextBuffer<Crypt, CharSet> Me;

template <class Crypt, class CharSet>
class UnixFilePagedTextBuffer: public TextBuffer<CharSet>,
protected UnixFile<Me, Crypt, CharSet> {
public:
    UnixFilePagedTextBuffer(std::basic_string fileName):
        TextBuffer<CharSet>(),
        UnixFile<Me, Crypt, CharSet>(fileName, this) {....}
    string getLine() {....read();....}
};
```



36

The text buffer depends on the character set at compile time, and on the output medium at run time. For example, the text buffer may take advantage of version information available in RCS files, or it may use permanent tags or line identifiers available in a database medium. We may want to write buffer contents to several different output media as the program executes. The output medium causes variation both in algorithm and structure of text buffers. We handle this using a variant of the multiple dispatch idiom. When the write member function of **TextBuffer<CharSet>** is invoked, it defers to its associated output medium (of unknown type) to dispatch to the proper write function. **OutputMedium** obliges (as in **UnixFile::write** above) by turning around and invoking the appropriate member function of **TextBuffer<CharSet>** (see above).

The variability in output type also drives a variability in structure; this is captured in the output medium class, rather than in the text buffer class itself. The algorithm specific to each pair of buffer types and output media appears in the member function (named for the output medium) of the corresponding class (named for the buffer type). Different derived classes of **TextBuffer<CharSet>** each have their individual implementations of **unixWrite**, **databaseWrite**, and other function specific to output media types.

Encryption; main

```
class RSA {
protected:
    void encrypt(string &);
    void decrypt(string &);
};

int main() {
    UnixFilePagedTextBuffer<RSA, wchar_t> buffer;
    string buf = buffer.getLine();
    ....
}
```



37

As stipulated by the domain analysis, an output medium is associated with a text buffer type at compile time (and with a corresponding text buffer instance at run time). Note that for this main program, no object code for **FullFileTextBuffer** will be incorporated in the executable, though object code for all output media will be present.

We can handle encryption with inheritance. It would be straightforward to derive a new class from **UnixFile<char, PagedTextBuffer<char> >**, overriding the encrypt member function, to create a new family member that supported an encryption algorithm of our choice. If the number of encryption algorithms is limited, they can be stockpiled in a procedure library and suitably called from the overridden encrypt function in the derived class. As another alternative, we can do things in a more “object-oriented way”.

Architecture Summary

- **The architecture should match the domains for good evolution**
- **Choose a solution binding suitable to each domain**
 - C++ covered here
 - Java, C# are also possible with limitations
- **Other advanced notions for free**
 - Formalizes the notion of paradigm in terms of commonality/variability pairs
 - Is essentially aspect-oriented design
 - Breakdowns in symmetry point to the need for patterns
- **Object Orientation is one common structure from domain analysis**
 - As with any paradigm, you still need engineering constructs—those are next



38

Multi-paradigm design finally provides a design method suitable for leveraging the C++ programming language. It is a general technique that avoids the pitfalls of using any single design method. It is in fact a form of meta-design useful for choosing the right paradigms. One still needs the tools and techniques of individual design styles, such as object orientation, to complete the design. And the technique is not mechanical or automatable; it is suggestive rather than prescriptive.

Because of its generality, multi-paradigm design can express concepts that elude object orientation. This leads to more maintainable designs, particularly for complex systems. It can capture the design rationale behind constructs such as overloading and templates in ways that OOD cannot and that UML certainly cannot.

4. Applying OO Fundamentals

- A. **Definitions**
 - > Inheritance
 - > Abstract Data Type
 - > Role
- B. **Classification Concepts**
- C. **Role-Based Modeling**
- D. **A Shapes example**

<i>Window</i>	
<i>Display Characters</i>	<i>Keyboard</i>
<i>Scroll Contents</i>	<i>View</i>
<i>Erase Contents</i>	<i>Mouse</i>
<i>Draw Lines</i>	

4A. Definitions — Quickly

- **Object**
- **Class**
- **Instantiation**
- **Substitutability**
- **Inheritance**
- **Polymorphism**
- **Abstract Data Type**
- **Role**



40

Next we will define some common terms for the sake of being able to communicate with each other. These definitions conform to common industry use.

4B. Classification Concepts

- **We can classify objects by their data structure**
 - The result is commonly called a *class*
- **We can classify objects by their behavior**
 - The result is a *role*
 - We may use a C++ abstract class or Java interface to express a role
 - An object might be of multiple types
- **We can classify classes by their behavior**
 - The result is an *abstract data type*



41

We can revisit many of these concepts in terms of classification, as above. The conceptualization space is richer than most object methods usually afford. Here, we carefully separate them. Why? It gives us more agility. Tying them together is like a three-legged sack race. We want separate roles to be able to run independently: Drawable should not necessarily be tied to Shape.

Inheritance

- **A relationship between classes—not types**
- **Not an analysis concept!**
- **An implementation mechanism used for:**
 - Polymorphism and subtyping
 - Code reuse
- **Supports substitutability**
- **Its main use is to organize software families in the architecture, a product of domain analysis**



42

Subtyping is a relationship between classes whereby one class takes on some implementation of another. The base class is the donor of such functionality; the derived class is the recipient. The derived class may override the member functions of the base class where it thinks it knows better how to implement such (e.g., class Circle can implement the rotate operation more optimally than its base class, Ellipse).

Inheritance is the usual mechanism to support subtyping in C++ object-oriented programming and also supports subtyping in Java. In a more vulgar sense, it supports code reuse. We will explore inheritance in more depth in section III.B below.

Abstract Data Type

- Or just “type”
- The “abstract” is a misnomer: it just means we take away the implementation
- We can talk about the complex ADT:
 - Fully definable in terms of behaviors
 - Complete and formal, without regard to how it’s implemented
 - A total classification
- **ComplexNumber is a type**
- **Type is behavior**
- **Abstract type is pure behavior**
- ☒ **The implementation of a type is a class**
 - ☒ *PolarComplexNumber is a class*



43

An abstract data type is abstract in the sense that it lacks code. The code that implements an ADT is called a class. Though there is no code, an ADT can be fully specified.

Role

- A ***partial*** interface to an object
- Therefore, a partial ADT
- A set of responsibilities related by some business concern
- E.g., Circle adheres to the roles Shape and Drawable
 - Shape: area, circumference, move
 - Drawable: draw, color, erase
- These are different domains!



• Roles offer a kind of flexibility beyond what is found in ordinary object designs or even in good domain design. This flexibility supports agile development during software maintenance embracing change.



44

A role is closely related to an ADT except we usually think of it as a partial classification rather than as a total classification. Like an ADT, it is a set of responsibilities related by some business concern. Here, we again separate out the concepts of Shape and Drawable from DrawableSquare.

We will use roles largely in conjunction with Use Cases: We will collect the related responsibilities of an Actor in a Use Case, and turn them into Roles.

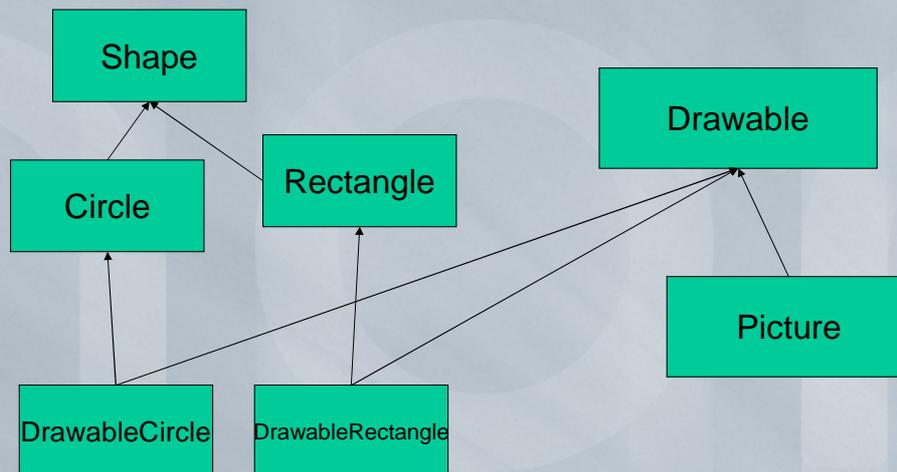
4C. Role-Based Modeling

- A role is *an* interface to an object
- Customers care much more about roles than about objects!
- ☒ An object may have many interfaces
- Remember, “interface” is the essence of an ADT, so roles are closely related to ADTs
- One can interpret the CRC card approach as a role-based approach
- Translation to C++ (ABCs) and Java (interfaces) is straightforward



• CRC cards give customer connection, avoid heavyweight documentation, and easily accommodate change on the frequency of minutes

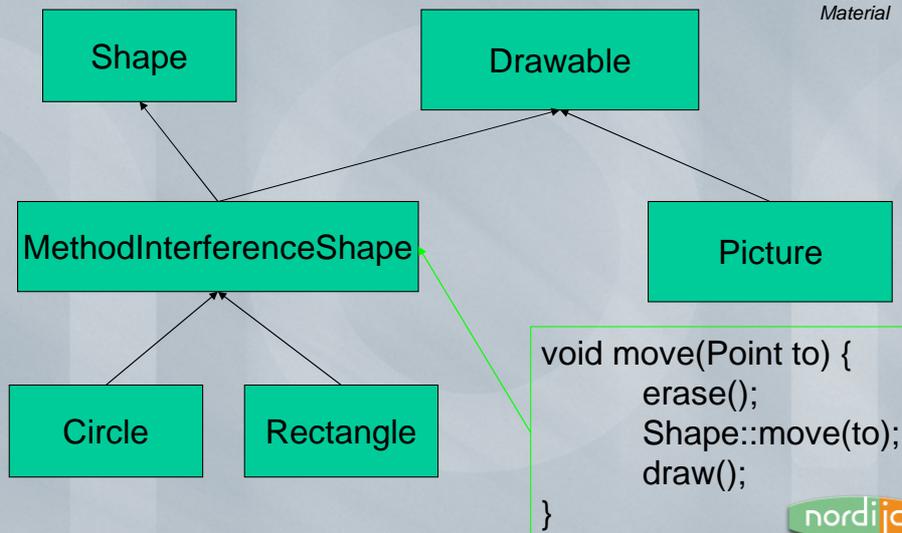
4D. An example: Shapes



Here is one implementation of a suitable Shape / Drawable hierarchy. We may have some common code at all levels, but it is surprising how much of the code falls into the derived classes.

Another Factoring of Shapes

Supplemental
Material



Here is another alternative, which perhaps a bit better factoring of commonality. However, with just this design, we could only have our GeoSat understand Circles if it also understood Drawables.

You need to do careful domain analysis to separate these out.

5. Responsibility-Driven Design

- Responsibilities tend to be stable over time.
- Contrast with functions, which change over time.
- Responsibilities: Relate to the problem domain.
- Functions: Relate to the structure of the solution.



• The outward focus of responsibilities is in line with focusing on individuals and interactions over focusing on processes and tools



48

We earlier discussed the importance of scenarios: that they help us identify the responsibilities of the objects in our system. Scenarios change over time; the responsibilities we seek should be stable over time. You may drive your automobile over many different courses, but the automobile supports you through each journey with the same set of responsibilities: accelerate when you depress the accelerator, stop when you depress the brake, and turn in the direction of the steering wheel.

Responsibilities are expectations, and we find these are stable over time in most domains. The functions that implement those expectations change with technology and with different customers. Your steering may be implemented as power steering, rack-and-pinion, or as a conventional universal joint. All three functions implement the same responsibility.

We will focus on responsibilities, and on the relationships between objects that those responsibilities imply in the context of the system's role in life. Responsibilities and relationships should be the focus of good system design. This holds for software systems as much as it does for sociological systems, where responsibilities and relationships are the keystones to sound structure.

“Who” is Responsible?

- We can identify responsibilities – things that need to be done.
- *Objects* are responsible for carrying out the responsibilities.
- Responsibilities defines the *role*.
- Roles are interfaces to *objects* of some *class*.
- *Classes* gives context to the responsibilities.
- We iterate the list of *classes* and their responsibilities, as well as the list of *roles* and their responsibilities.



49

Focusing on responsibilities alone isn't enough: there is also the issue of accountability, of where the responsibility lies. We attach responsibilities to objects. Objects become the agents, the experts, the specialists who work together inside our software systems to achieve the overall system goal. They do so by being faithful to their responsibilities. The system works because the responsibilities—the services available in the “society” of our program—support the use cases we expect of the system.

Responsibilities define the classes. If you are a manager building a team, you characterize the roles to be filled in terms of the responsibilities you expect of those roles. The role is defined by its responsibilities.

But classes define the responsibilities as well. Existing libraries and conventions suggest or impose mappings of responsibilities onto objects. The knowledge and data inside a class suggest some of its responsibilities, or dictate where the designer should assign responsibilities.

Both views must be considered, and matching responsibilities with classes is usually an interactive task. Use cases help move the iteration along.

CRC Cards: Classes, Responsibilities, and Collaborators

<i>Window</i>	
<i>Display Characters</i>	<i>Keyboard</i>
<i>Scroll Contents</i>	<i>View</i>
<i>Erase Contents</i>	<i>Mouse</i>
<i>Draw Lines</i>	



50

The tool we will use for object-oriented analysis is a **CRC Card**. There is nothing special about a CRC card: it is just a 8cm x 13cm (or perhaps slightly larger) lined paper index card. On the very top, we put the name of the class (here, it is Window). On the left side of the card, we list the class responsibilities (Display Characters and so forth). On the right side, we list the helpers (Keyboard, View, and Mouse) that will work together with this class to fulfill its responsibilities. Helpers are other classes in the system.

The size of the card is important. We want to write succinct, semantically rich, concise words for responsibilities and class names. We don't want you to use these cards to write an FSD! The cards are not important: what is important is the team interactions and team understanding they bring out during a design exercise. So if you lose your deck of CRC cards, don't worry—the important design information is in your head, and in the heads of your teammates.

CRC cards are an **informal** tool. Don't try to line up collaborators on the same lines as the responsibilities. Keep them small. Keep them simple. We will be discussing many “rules of thumb” for CRC cards during the rest of the day.

CRC Cards

- Carry out scenarios.
- Done by a group of domain experts.
- Little object-oriented expertise necessary.
- Great team-building tool.
- Great for building shared vision of architecture.
- Great to identify stake-holding relationships.
- ▶ Something no CASE environment can do.



- The Organizational Patterns speak of GROUP VALIDATION.
- This is a form of Test-Driven Development
- The Manifesto: Individuals and Interactions over Process and Tools


51

CRC cards are usually filled out during a role-playing meeting. The participants should be a small team of domain experts. This team will be the one that shapes the project (or subsystem). It is not only important that the right domain expertise is represented, but that the individuals eventually form a cohesive group where motivations and insights are shared and understood.

It is more important that these analysts understand the application domain than objects. The CRC technique relies on the intuitions of the domain experts present more so than on any notion of objects. If there is a facilitator present, CRC cards can be used by a team with little or no prior exposure to them. It is important that an experienced facilitator oversee the first couple of meetings to help the group learn the technique.

The primary output of a CRC meeting is a deck of cards. But more important than that, the meeting develops understanding of why this set of cards was selected, and why responsibilities were allocated as they were. **It builds a shared vision of the system architecture across the entire development team.** Furthermore, it helps team members identify stake-holding relationships, sources of expertise, and the orientation of other team members to the project. **It is a great team-building exercise.**

Organizations using CASE environments as their primary design tool rarely reap these benefits. You can't develop the same completeness of design understanding by sitting at a work station drawing bubbles and arrows as you can socializing a design in an interactive meeting.

CRC: The Social Setting

- **JAD-like: make sure stakeholders are there or represented**
- **Let specialists play roles**
- **Two modes:**
 - All cards on the table, arranged according to their coupling
 - “Create” cards as they are needed
 - Each person owns one or more cards (Standish report encourages ownership)



Manifesto: Customer Collaboration over Contract Negotiation. Have all your stakeholders there.



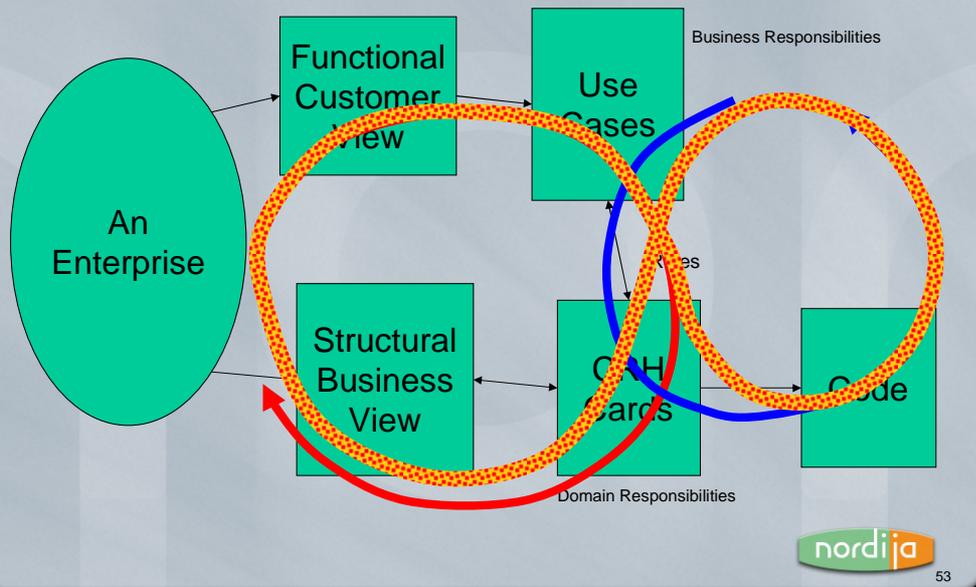
52

CRC cards are reminiscent of Joint Application Development: a design technique based on extensive stakeholder presence and engagement. There are two ways to get it started: either by letting each stakeholder own cards, or by letting a group stand around a table on which the cards are arranged. It is O.K. (and sometimes necessary) for a given individual to “own” one or more cards.

The Standish report on software cost overruns (1994-1995) established that ownership is a significant factor in software cost; this is a good place to start identifying ownership relationships.

6. Tying it Together

Investment pays off



Now we want to tie everything together. By dividing analysis into domain analysis of long-term stable assets, and Use Case analysis of evolving user functionality, we have a complete picture of what needs to be built. Now we need to reconcile those two views with each other, to assure that the system will work (to the degree we can) and to sling code.

In summary, we will extract roles from the Use Case actors, make sure that the roles fit the classes in the Domain Analysis, all using the interactive social technique of CRC cards. Once we can “run” the exercise using CRC cards, we can code it up.

Design Alternatives

- Do domain analysis; use CRC cards to create roles; then group classes that are covered by the role, iterating as necessary; implement classes that interact through roles
- Use CRC cards to identify classes as loci of responsibilities; iterate; implement
- Use CRC cards to identify objects as loci of responsibilities; distill objects into classes; iterate; implement

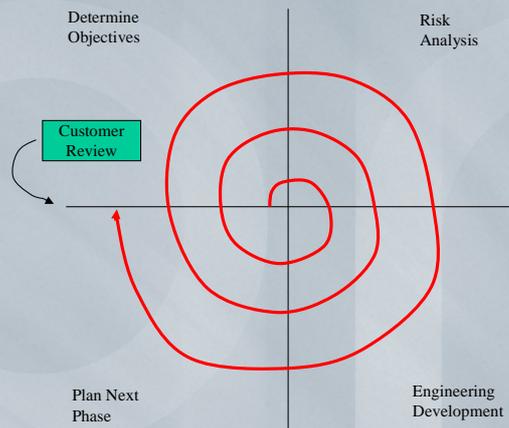


54

There are several ways to skin a cat; some common alternatives are described above. Exactly how you approach this will depend on business needs and on your culture.

The Big Process: Spiral

- “Evolutionary waterfall”
- Get information on only the most important features first
- Design, test, implement those
- Test using Use-Case-derived tests
- Get customer feedback
- Adjust, and incorporate next level of detail
- Add new Use Cases as necessary
- Each step is a small waterfall, with the ability to skip steps



nordija

55

The outer, enclosing process is fundamentally iterative. It is good to add structure to the process. If you are doing SCRUM, you already have such structure in place. SCRUM is a form of spiral, as shown above.

Factoring and Refactoring

- **Over the long term, you find more common code**
 - Expansion of the Domain
 - Commonalities across customer sub-communities
 - Push such code into the domain
- **Over time, you find better ways to do things**
 - Do refactoring after a delivery when you are in a lull preparing for the next project
 - Refactor with discipline. Refactoring without using refactoring tools is asking for trouble.
 - Do not refactor at overly fine levels of granularity: you end with local optimization — finest level where systems thinking is possible
 - Refactoring is not “license-to-hack”
 - Maintain the domain structure!



56

7. Implementing the Design

- Mapping the CRC cards into code
- Missing Responsibilities
- Instantiation
- Example Code
- Engineering Practices
- Back to Agile
 - Use Cases
 - The place of architecture

This is an outline of the final section

Mapping the CRC cards into Code

- **Responsibilities become member function declarations on roles**
- **Use collaborators to help discover arguments**
- **Each role should “wrap” part of a class**
 - If every role responsibility is a class responsibility, then just access the instantiated object through the role
 - If a class misses some role responsibilities, then
 - Re-factor the design, OR
 - Fullfill the role with an aggregate of objects, OR
 - Distribute the responsibility across multiple objects
- **See Following Slides**



58

Special case: Instantiation

- **Normally, we just create an object and associate it with a pointer (in C++) or other identifier**
- **When a role subscriber instantiates an object, it must instantiate the actual object behind the interface**
 - Declare identifier in terms of the role
 - Create object in terms of the domain class

C++ Example

```
#include <rope>
rope<char> fileBuffer ; // in-core file content

class stringInterface { // role
    const virtual char operator*(void) const = 0 ;
    virtual stringInterface &operator++(void) = 0 ;
    virtual char &operator[](int) = 0 ;
};

UnixFile *fp = new UnixFile<textBuffer>(file, tb) ;

fileBuffer = fp->wholeFileAsRope() ; // get content
stringInterface *file = &fileBuffer ; // role

stringInterface *textBuffer = // another role
    new UnixFilePagedTextBuffer<trapdoor> ;

....

/*
 * Now copy role-to-role
 */
while (*textBuffer++ = *file++) { }
```



60

In this example we eventually want to copy characters between two objects using a string-like protocol. We define `stringInterface` as the role that elicits the string behavior of a number of string-like objects. We “wrap” each of a `rope<char>` and `UnixFilePagedTextBuffer` with a `stringInterface` role. Now we can treat them each as string-like objects without regard to their underlying form.

If at some point we want to change the code to use `std::string<char>` instead of `rope<char>`, or if we want to move the copying code in to a more broadly used character-copy function, the role protocols insulate the copying code from those changes.

Java Example

```
interface Fruit {
    public boolean isCitrus();
}

interface Vegetable {
    public boolean isARoot();
}

class Tomato implements Fruit, Vegetable {
    boolean citrus = false;
    boolean root = false;
    public Tomato() {}
    public boolean isCitrus() { return citrus; }
    public boolean isARoot() { return root; }
}

....

public static void main(String[] args) {
    Tomato tomato = new Tomato();
    Fruit tomatoFruit = tomato;
    Fruit orange = new Orange();
    Vegetable tomatoVegetable = tomato;
    Vegetable agurk = new Agurk();
    ....
}
```

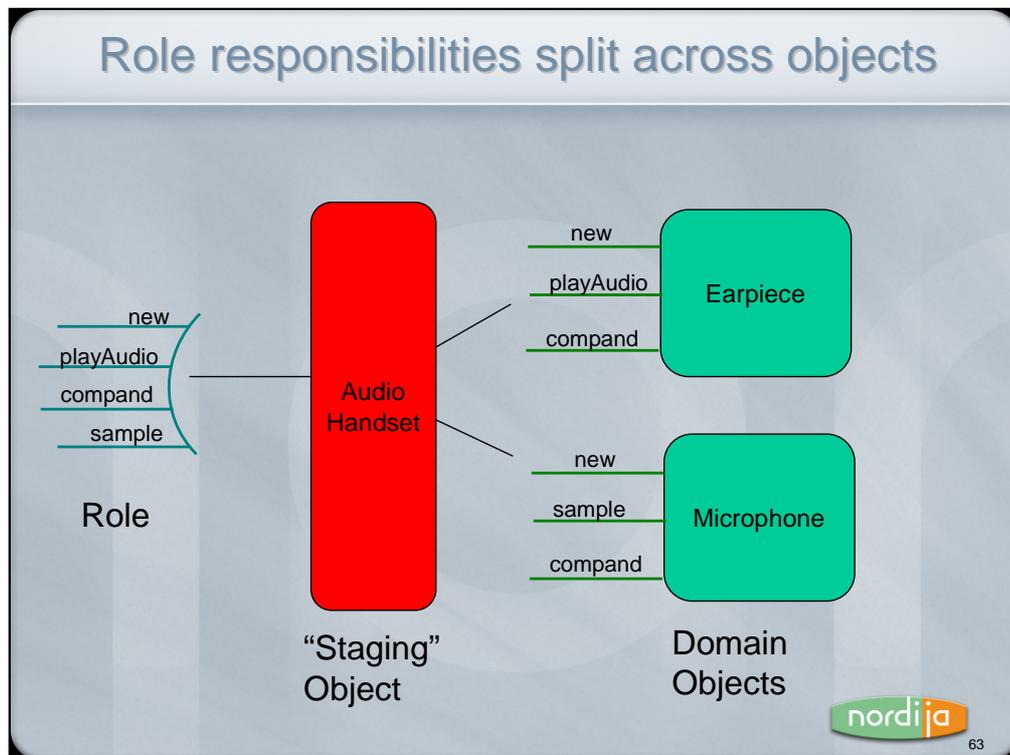


61

Here is a trivial Java example, adapted from <http://www.fluffycat.com/Java/Interfaces/>.

Missing responsibilities?

- **What if your Use Case has a responsibility that is not in the domain model?**
 - E.g., “Reset System” → Return to the domain model and iterate
- **What if one role’s responsibilities are split across several objects?**
 - Create a new domain object that satisfies the role
 - Create a class for that object that instantiates (or connects to) all objects necessary to satisfy the role’s interface
 - An instance (object) of that class will act as the role



If a CRC card has responsibilities that can't all be found on one domain class, you need to create a community of objects that together fill the role. Create a new domain class so you can instantiate a staging object that creates instances of the domain objects as needed. The staging object can be used where a role (Java interface or C++ abstract base class) would otherwise be used.

Don't forget good engineering practices

- **Laws of Demeter**
- **Pre-conditions and post-conditions**
- **Inheritance and subtyping rules**
- **Covariance and contravariance**



• 50% of development effort lies in understanding the code: the *discovery problem*. Good engineering practices reduce discovery costs and lead to agility



64

So far, we have just done analysis and basic design. Good programming requires some engineering skills. Object orientation comes with a wide palette of engineering techniques. The Laws of Demeter help you audit coupling and cohesion, and further insulate domain classes from changes in customer whims. Pre-conditions and post-conditions can help you gauge the correctness of inheritance and, in general, the adherence of the design to requirements that come from Use Cases. Covariance and contravariance are unusual problems that must be understood when building maintainable inheritance hierarchies.

8. Course Summary

- **Architecture is important**
 - Analyze your domain to save costs
 - Analyze your customers to generate revenues
- **Design**
 - Solving the problems of requirements and costs
 - In general, you need multiple paradigms
- **An object**
 - Encapsulates key domain knowledge
 - Provides an interface that can take on many roles
- **Classes and inheritance**
 - Important engineering and implementation concerns
 - Effective use of subtyping for architectural expressiveness
- **CRC cards bring it all together**



65

References

- Coplien, J. *Multi-paradigm Design for C++*. Reading, MA: Addison-Wesley, 1998.
- Coplien, J. *Multi-paradigm Design for C++*.
<http://users.rcn.com/jcoplien/Mpd/Thesis/Thesis.pdf>
- Reenskaug, Trygve. *Working with Objects: The OOram Software Engineering Method*. Greenwich: Manning Publications, 1996.