Object-Oriented Frameworks

Greg Butler
Department of Computer Science
Concordia University, Montreal

gregb@cs.concordia.ca
http://www.cs.concordia.ca/~faculty/gregb

Abstract

Frameworks offer a concrete realization of a product line. A framework is an architecture, plus an implementation, plus documentation that capture the intended use of the framework for building applications. A framework provides a highly effective mechanism for software reuse within an application domain. The framework captures the features that are common across the product line. In return for relinquishing some design authority, the developer can build a new application faster by hooking to the framework just the code that is unique to the new application.

The tutorial presents methodologies for the development, application, and evolution of object-oriented frameworks. Concepts and techniques behind modeling and implementation of the commonality and variability within a domain are presented.

Level: Intermediate

Required Knowledge: objects, polymorphism, delegation, composition; some design patterns.

Outline

- What is a Framework? What it is not!
- Framework Development, Application, Evolution.
- Development Concepts,Techniques and Models
- Documentation for Application Developers
- Wrap-up, Questions, Open Issues.

Outline

- **What is a Framework? What it is not!**

- Framework Development, Application, Evolution.

- Development Concepts,Techniques and Models

- Documentation for Application Developers

- Wrap-up, Questions, Open Issues.

What is a Framework? What it is not!

- What is a Framework?
- Basic Properties of Frameworks
- Frameworks vs Libraries
- A Toy Example
- Some Real-World Examples

- Concepts
- What a Framework is Not!
  - architecture, design pattern, domain analysis
  - product line

What is a Framework?

… a collection of abstract classes, and their associated algorithms, constitute a kind of framework into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. The framework consists of the abstract classes, the operations they implement, and the expectations placed upon the concrete subclasses.[Deutsch, 1983]

What is a Framework?

A framework is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific. [Johnson and Foote, 1988]

What is a Framework?

… *a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.* [Gamma et al, 1995]

What is a Framework?

Common points

- framework addresses a domain/product family
- framework prescribes how to decompose a problem
- design of an application or subsystem
- set of classes and how they collaborate
  - shared invariants of objects and how to maintain them
  - conform to model of concepts and collaborations
- i.e., framework is represented by its code
- use a framework to build applications by
  - creating new subclasses
  - configuring objects together
  - modifying working examples

What is a Domain?

A *domain* is an area of knowledge that is

- scoped to maximize the satisfaction of the requirements of its stakeholders,
- includes a set of concepts and terminology understood by practitioners in that area, and
- includes the knowledge of how to build software systems in that area.

Basic Properties of a Framework

- Modularity: abstract classes with stable interfaces, encapsulating and localizing change in ``hotspots'' (= points of planned variability)
- Reusability: of analysis, design, and code
- Extensibility: by providing explicit hotspots or ``hooks'' for planned variability

- Inversion of Control: *Don't call us, we'll call you.* (The Hollywood Principle)

Frameworks vs Libraries

Library case

Application developer writes a main program, determines flow of control and problem decomposition. Custom code calls the library code.

Framework Case

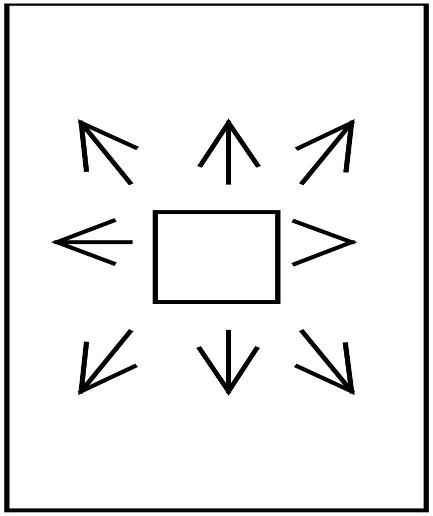
Application developer writes a subclass. Framework has determined flow of control and problem decomposition already. Framework code calls custom code.

Goals of Frameworks

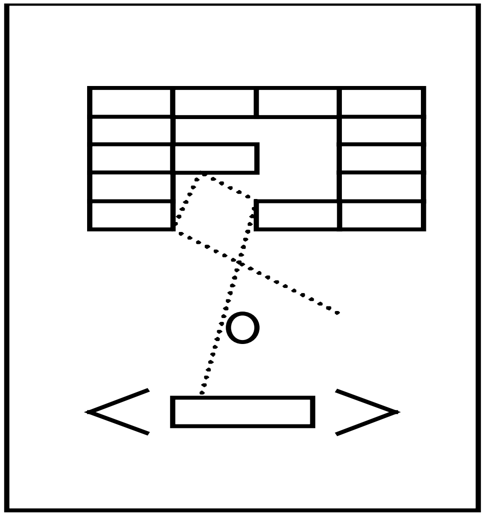
- Make it easy to develop applications.
- Write as little new code as possible.
- Enable novice programmers to write good programs.
- Leverage domain experience of expert programmers.

A Toy Example

Domain of Bouncing-Bumping Games



Bump-Em Car



Brick World

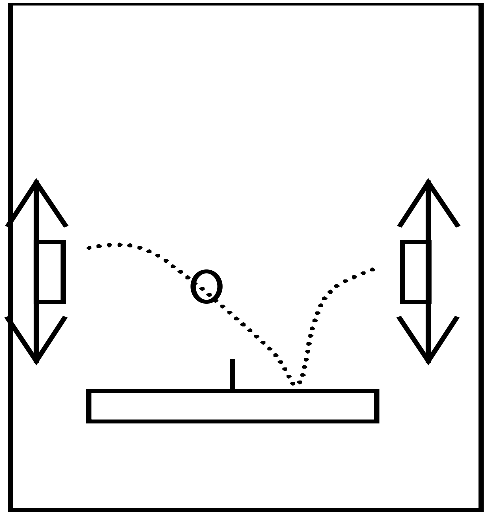
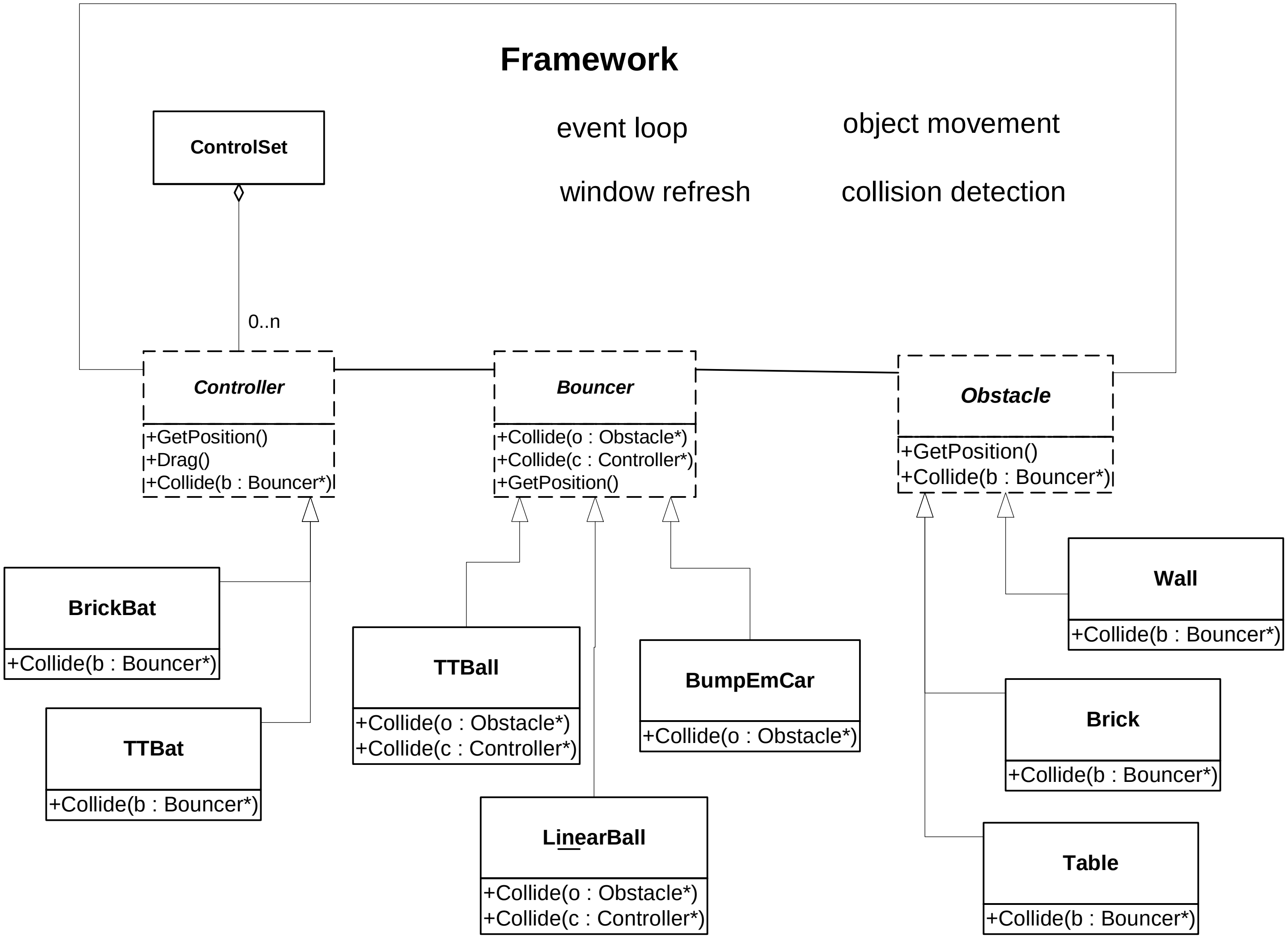


Table Tennis


Framework
ControlSet
event loop
object movement
window refresh
collision detection
0..n
Controller
+GetPosition()
+Drag()
+Collide(b : Bouncer*)
Bouncer
+Collide(o : Obstacle*)
+Collide(c : Controller*)
+GetPosition()
Obstacle
+GetPosition()
+Collide(b : Bouncer*)
Wall
+Collide(b : Bouncer*)
BrickBat
+Collide(b : Bouncer*)
TTBall
+Collide(o : Obstacle*)
+Collide(c : Controller*)
BumpEmCar
+Collide(o : Obstacle*)
Brick
+Collide(b : Bouncer*)
TTBat
+Collide(b : Bouncer*)
LinearBall
+Collide(o : Obstacle*)
+Collide(c : Controller*)
Table
+Collide(b : Bouncer*)

Template Method and Hook Method

These are the basic building blocks for commonality and variability in code.

A *template method* provides the generic algorithm or steps for a task. It calls one or more *hook methods*.

Each *hook method* represents a point of variability by providing the calling interface to a variable task. Each implementation of a hook method provides a variant of that task.

A Toy Example

template methods

```
Game::makeWorld(){
    makeBouncer();
    makeControllers();
    makeObstacles();
    makeEventHandlerTable();
}

Game::Run(){
    loop over event e in eventQueue{
        ehTable[e]->handleEvent(e);
        refreshDisplay();
    }
}
```

hook and template method

```
TimeEventHandler::handleEvent(){
    bouncer->update Position();
    loop over detected collisions <b,o> {
        b->Collide(o); //bouncer
        o->Collide(b); //control or obstacle
    }
}

ControlEventHandler::handleEvent(){
    c->updatePosition();
}
```

Examples of Frameworks

GUI Examples

- Taligent/IBM OpenClass
- Microsoft MFC, Borland OWL, Java Swing
- MET++ *(Phillipe Ackermann et al)* Multimedia

Business Examples

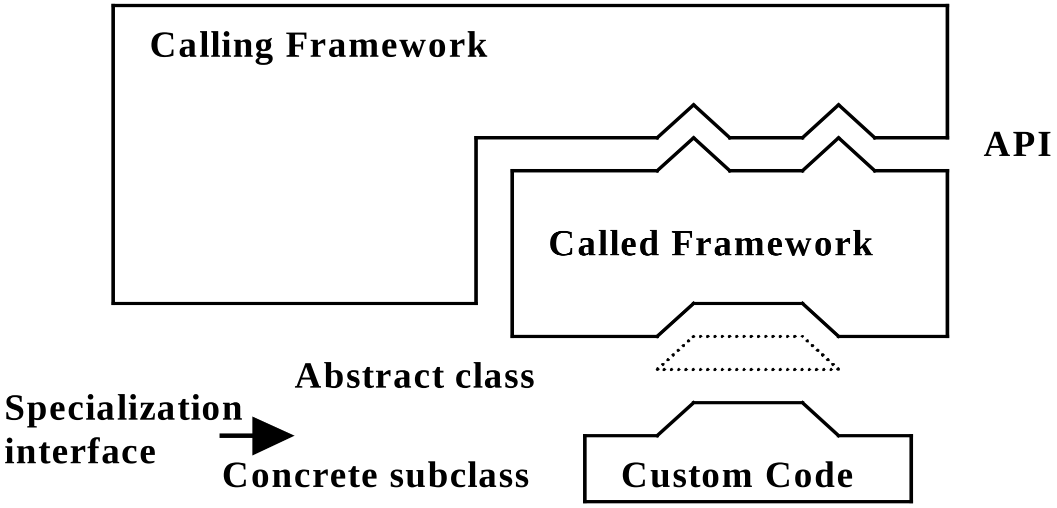
- SAP, Baan, IBM San Francisco business objects

Other Examples

- Adaptive Communication Environment *(Doug Schmidt et al)*
- CelsiusTech ship warfare systems
- Tektronix oscilloscope software
- Hewlett-Packard laser printers

Some Concepts

- subframework
- specialization interface = the hook metthods of the abstract class
- commonality-variability
- customization



Frameworks vs Software Architectures

Architectures

- are design artifacts
- can be for single applications, not just for product lines
- are designed with many qualities in mind

Frameworks are *concrete* implementations of *semi-complete* architectures.

Frameworks are designed to be

- reusable
- specialized

Frameworks vs Design Patterns

Design patterns identify, name, and abstract common themes in object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities.

Design patterns describe micro-architectures.

Design patterns are abstract.

Framework has a concrete architecture.

Framework design may incorporate design patterns at the micro-architectural level.

Flexibility for specialization of a framework is often provided by a design pattern.

Frameworks vs Domain Analysis

Domain analysis is a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems.

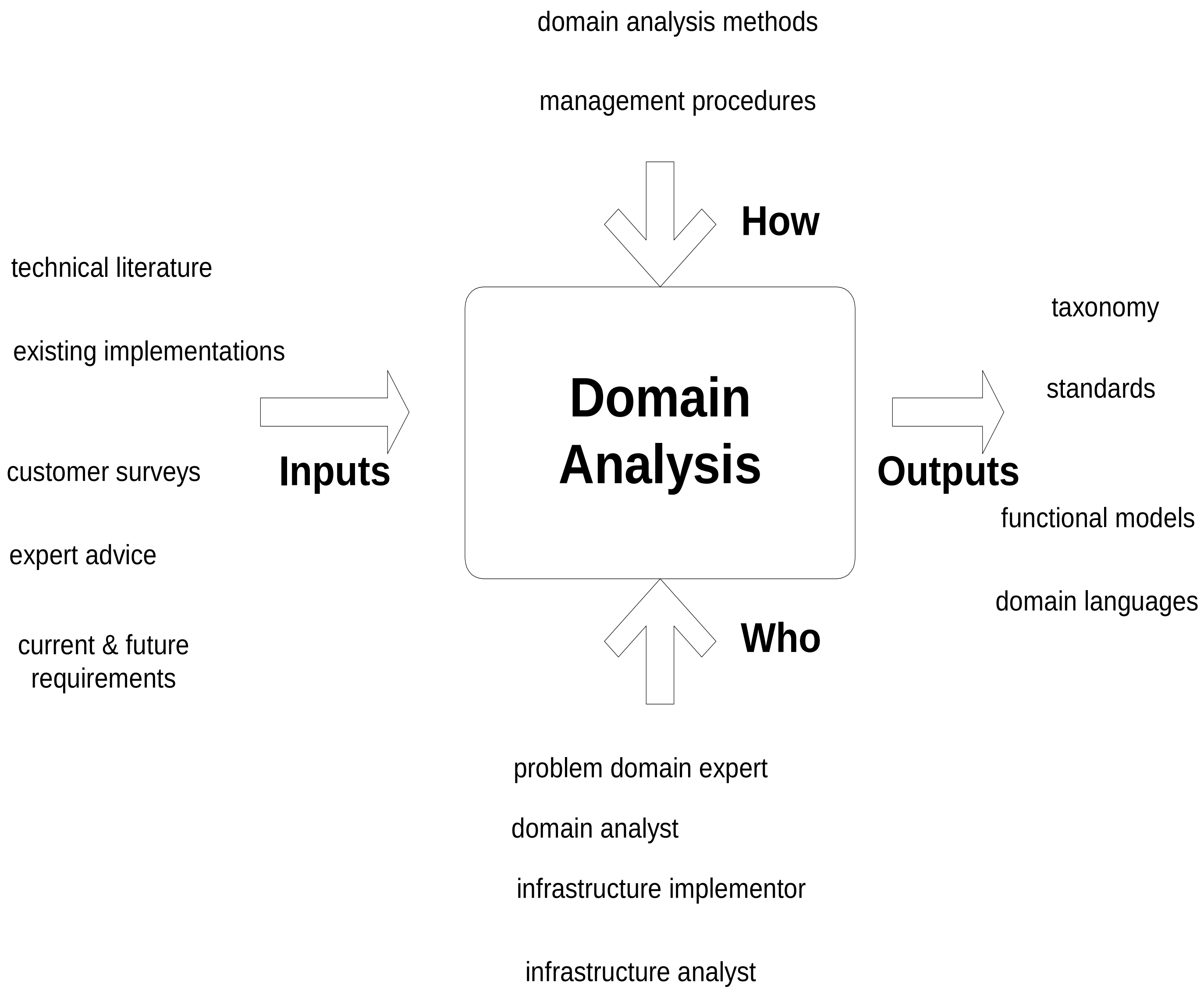
Domain analysis is concerned with a family of products

- **problem domain,** i.e. context and requirements
- **solution domain,** i.e. applications

Frameworks focus on *solution domain*

Framework is instance of a domain specific software architecture (DSSA)

Overview of Domain Analysis



domain analysis methods

management procedures

How

technical literature

existing implementations

customer surveys

expert advice

current & future
requirements

Inputs

Domain Analysis

Outputs

taxonomy

standards

functional models

domain languages

Who

problem domain expert

domain analyst

infrastructure implementor

infrastructure analyst

Frameworks vs Product Lines

A *product line* is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market. [Withey, SEI, 1996]

Product line is market-oriented, with feature set dominant.

Several ways to implement product lines.

Product line may have multiple architectures.

Framework has solution domain (= code) focus.

Framework is an implementation of a product line.

Framework has only one architecture (almost always).

OK to say ``framework = product line’’

Outline

- What is a Framework? What it is not!
- **Framework Development, Application, Evolution**
- Development Concepts,Techniques and Models
- Documentation for Application Developers
- Wrap-up, Questions, Open Issues.

Development, Application, Evolution

- The Different Uses of a Framework
  - Development
  - Application
  - Evolution

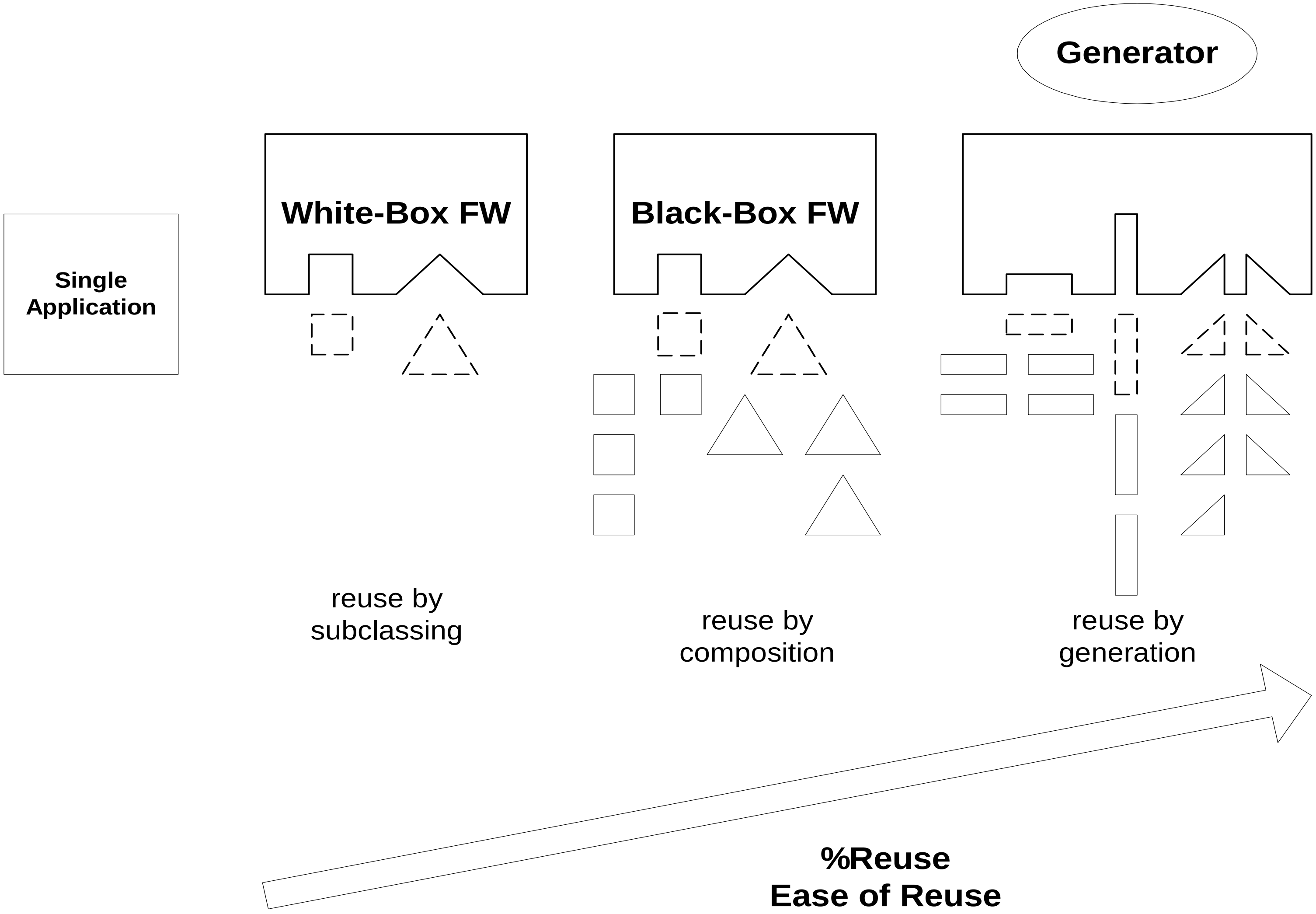
- Framework Maturity Lifecycle
  - white-box to black-box to visual builder

- Reuse Tasks with a Framework

The Different Uses of a Framework

- Development of a Framework
  - create the infrastructure for a product family
  - by domain experts
- Customize Framework to Develop an Application
  - the most important case, done many times
  - by novice programmers
- Evolve the Framework
  - evolve the infrastructure

Framework Maturity Lifecycle



Generator

Single Application

White-Box FW

Black-Box FW

reuse by subclassing

reuse by composition

reuse by generation

%Reuse
Ease of Reuse

Reuse Tasks with a Framework

- **Selecting** a framework for the intended application

- **Composing** an object from a library of concrete subclasses (planned customization)
- **Extending** a hook in planned ways
- **Flexing** a hook by extending it in unplanned ways

- **Evolving** a framework to add new hooks or new flexibility to an existing hook
- **Mining** a framework for ideas applicable to a new context/domain

Outline

- What is a Framework? What it is not!
- Framework Development, Application, Evolution.
- **Development Concepts,Techniques and Models**
- Documentation for Application Developers
- Wrap-up, Questions, Open Issues.

Concepts,Techniques and Models

- Overview of Methodologies
- Example-Driven, Bottom-Up Development
  - refactoring
- Hotspot Generalization

- Modeling Commonality and Variability
  - feature model
  - use case model and variation points
  - design patterns, roles, templates, configuration
  - code templates and hooks

Framework Development Approaches

- Example-driven, Bottom-up, Incremental
- Top-down Domain Engineering
- Hotspot generalization
- Use case-driven
  - Catalysis
  - RSEB
  - Feature RSEB
- Hybrid

Some Variability Mechanisms

Mechanism	Type of Variation Point	Type of Variant	Use When
Inheritance	Virtual operation	Subclass	Specializing and adding operations
Extensions	Extension point	Extension	Attaching several variants
Uses	Use point	Use case	Reusing abstract use case
Configuration	Configuration item slot	Configuration item	Choosing alternative function or implementation
Parameters	Parameter	Bound Parameter	Selecting between alternate features
Template instantiation	Template parameter	Template instance	Type adaptation or selecti alternative pieces of code
Generation	Parameter or language scripting	Bound parameter or expression	Doing large scale creatio one or more classes

Feature, Hook, Hotspot

A *feature* is any aspect of a system used to characterize it to a stakeholder.

A *hook* is a point in the framework that is meant to be adapted in some way, such as by filling in parameters or creating subclasses.

A *hot spot* is a variable aspect of a framework domain, whereas a fixed aspect is called a *frozen spot*.

The **same concept**: different communities, different names!

Hook Descriptions

Name

Requirements Textual description of the problem that the hook is meant to solve.

Type

- **Method of Adoption:** *enabling*, *disabling*, *replacing*, *augmenting*, *or adding* a feature
- **Level of support:** *option*, *supported pattern*, *open-ended*

Area The parts of the framework affected by the hook.

Uses The other hooks that use this hook.

Participants The components that participate in the hook: both existing and new components.

Changes This is the main section that outlines the changes necessary to the interfaces, associations, and control flow amongst the participants.

Constraints

Comments

Bottom-Up Framework Development

Build one application at a time, evolve framework

Step 0: Gain lots of experience in domain

Step 1: Build first application (with domain analysis)

Step 2: Iterate

- 2.1 Change impact analysis for n-th application
- 2.2 Refactor existing framework to accommodate changes
- 2.3 Build n-th application

See [Roberts and Johnson, 1998]

Top-Down Framework Development

Plan all applications at once

Step 1: Domain analysis

Step 2: Develop DSSA (domain-specific software architecture)

Step 3: Implement DSSA

Step 4: Populate DSSA as applications are built

See [Kang et al, 1998]

Hotspot Generalization

Plan all applications at once

Step 1: Identify hotspots

Step 2: For each hotspot

- 2.1 Classify flexibility required (as a meta-pattern)
- 2.2 Associate it with a subsystem/façade class
  - select a design pattern for required flexibility
  - the façade class is composite if there are multiple dimensions of variability within hotspot

See Chapters 15 & 16 in [Fayad, 1999]

Hotspot Descriptions

Name

Description

Common Responsibility

Concrete Examples of alternative realizations of the variable aspect.

Variability: Kind of flexibility required.

- **Parameters:** Whether some, or all, of the variability can be covered by parameterization.

Granularity: atomic or composite. For composite hot spots describe the dimensions. Give reasons why a composite hot spot is treated as atomic.

Multiplicity: number (1 or n) of variants bound to this hotspot. For n, whether the are *chain-structured* or *tree-structured*.

Binding Time: *application creation; runtime*, *once* or *many times*.

May also indicate whether binding/adaptation should occur *with* or *without restart* of the application.

Hotspot Meta-Patterns

Pree classified patterns based on the relationship between the *template method* t and the *hook method* h. He called the classifications ‘‘*meta-patterns*’’.

Unification means that t and h are in the same class TH.

Separation means that t is in class T, h is in class H, and there is an association from T to H.

Recursive means that class T is a subclass of class H, and there is an association from T to H. The *multiplicity* of this association can be 1:1 or 1:n.

T may be a direct or indirect subclass of H.

The degenerate case of recursive is when T is H (ie unification).

Meta-Patterns to Design Patterns

non-recursive hot spot design patterns

- Interface inheritance, Abstract factory, Buidler, Factory method, Prototype, Adapter, Bridge, Proxy, Command, Iterator, Mediator, Observer, State, Strategy, Template method, Visitor

1:1 recursive (= *chain*) hot spot design patterns

- Chain of responsibility, Decorator

1:n recursive (= *tree*) hotspot design patterns

- Composite, Interpreter

Use Case-Driven Development

Plan all applications at once, essentially classic OO

[Step 0: Domain analysis to get feature model]

Step 1: Capture functionality as use cases; model variability using *variation points* or *generalization*

Step 2: Map variation points to design patterns

See [Jacobson et al, 1997]

Domain Analysis Products

Basic products are:

- Context, scope, and boundary of domain
- Taxonomy/glossary/data dictionary
- Feature model
- Functional model, eg use cases
- Domain specific software architecture

Appendices: Information Sources, Examplars, Standards

SEI Product Line Guide adds perspective for each stakeholder

Hybrid Framework Development

Plan several applications at once, evolve framework

Step 1: Interleave

- 1.1 Do partial domain analysis
- 1.2 Do change impact analysis for new applications
- 1.3 Refactor existing framework to accommodate changes
- 1.4 Build new applications

Example-Driven, Bottom-Up Development

Principles

- Frameworks are abstractions: people generalize from concrete examples

- Designing reusable code requires iteration

- Frameworks encode domain knowledge

Frameworks Encode Domain Knowledge

- Understand domain, its problems, and examplar solutions
  - different viewpoints for different stakeholders
  - explain current design in terms of domain issues
- Separate *technology* frameworks from *application* frameworks
  - technology = GUI, object persistence, …
  - application = business domain
- Iteration is needed as domain concepts become better understood

Generalize from Concrete Examples

- People think concretely, not abstractly
- Abstractions are found bottom-up by looking at concrete exemplar solutions
- Generalization proceeds
  - identifying two things with different names that are really the same thing
  - parameterizing to eliminate differences
  - decomposing into parts, so similar components can be identified

Frameworks Require Iteration

- Getting domain right requires iteration
- Getting abstractions right requires iteration
- Reusable code requires many iterations

Law: Software is not reusable until it has been used in a context other than its initial context

Rule of Three

- Law requires three uses in other contexts
- Need three examplars before designing framework

Refactoring

Refactoring is a restructuring of a program that preserves behaviour.

- Reorganize inheritance hierarchy

- Move methods around
- Delegate method to implementation class

- Introduce design patterns

See Martin Fowler, Refactoring, Addison-Wesley, 1999.

Design Patterns for Variability

What Varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interaction between objects	Mediator
Object being created	Factory, Prototype
Structure being created	Builder
Traversal algorithm	Iterator
Object interfaces	Adapter
Object behaviour	Decorator, State

Feature Model

A *feature* is any aspect of a system used to characterize it to a stakeholder.

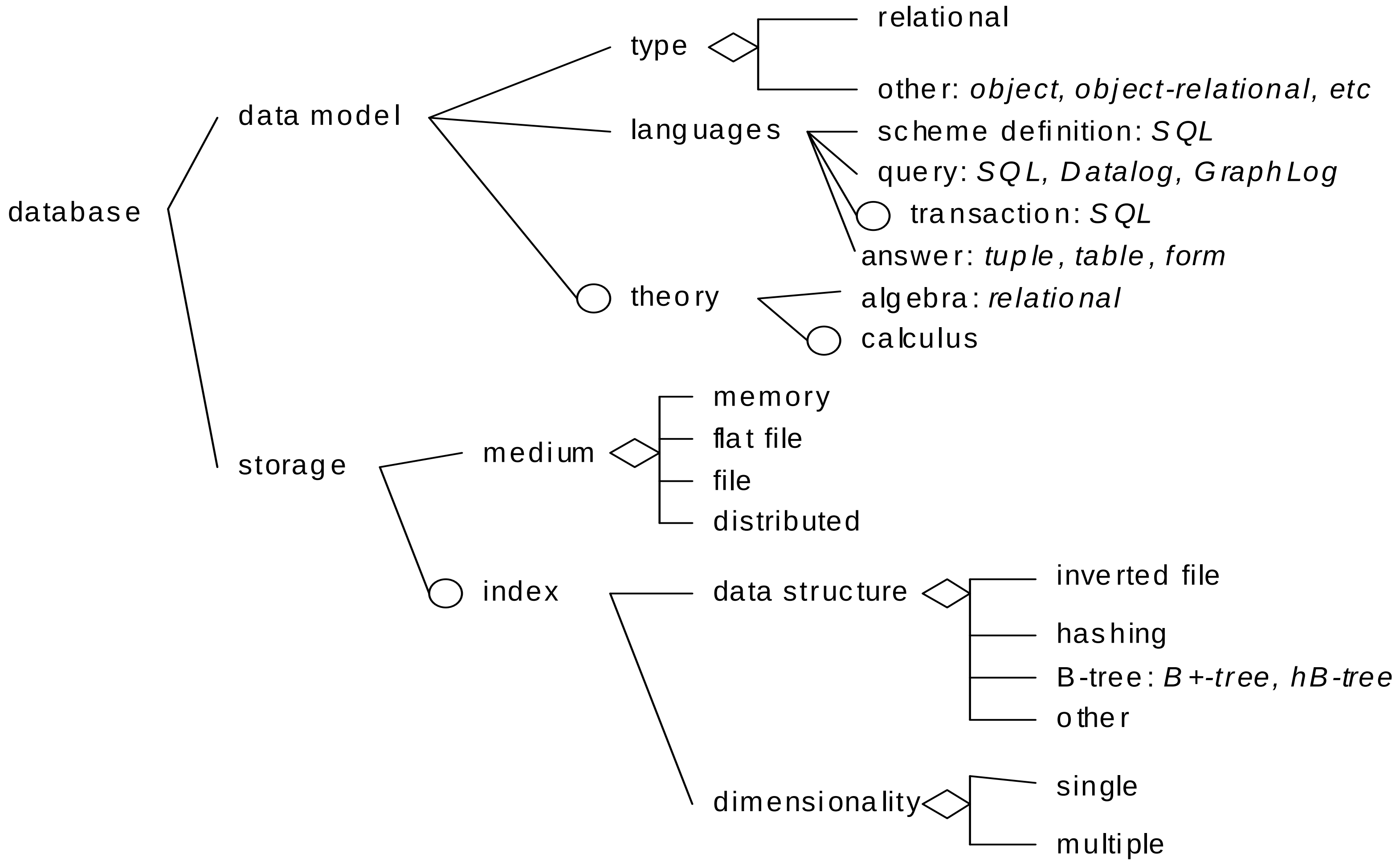
A *feature model* is the collection of all features and their relationships.

Each feature is classified as

- *mandatory*,
- *optional*, or
- *alternative*.

An application is determined by its *feature set*.

Example Feature Model



Feature Model

The relationships between features are:

- *aggregation*
- *generalization*
- *implemented-by*

and constraints can be put on feature sets

FORM **Feature Model**

The features can be categorized into layers:

- *Capability*: user-level functionality
- *Technology*: domain-specific techniques
- *Environment*: operating environment
- *Implementation*: languages, libraries, ...

Layered Feature Model

Feature Model	Feature Categories	Object Categories	Object Examples
Capability	Service	Service-state-hiding object	
	Operation	User-role object	
	Non-functional characteristic		
Operating Environment	Hardware	Interface	
	Software	Interface	
Domain Technology	Methods	Computational object	
	Standard/Law		
	Theory	Computational object	
Implementation Technique	Design decision	Computational object	
	Communication	Connector	
	Component/ADT	Entity	

Use Case Model

A *use case* is a set of cohesive interactions between actors and the system that performs a task of interest to the initiating actor.

A *variation point* in a use case is a ``hook'' for system actions or interactions within the template of the use case itself.

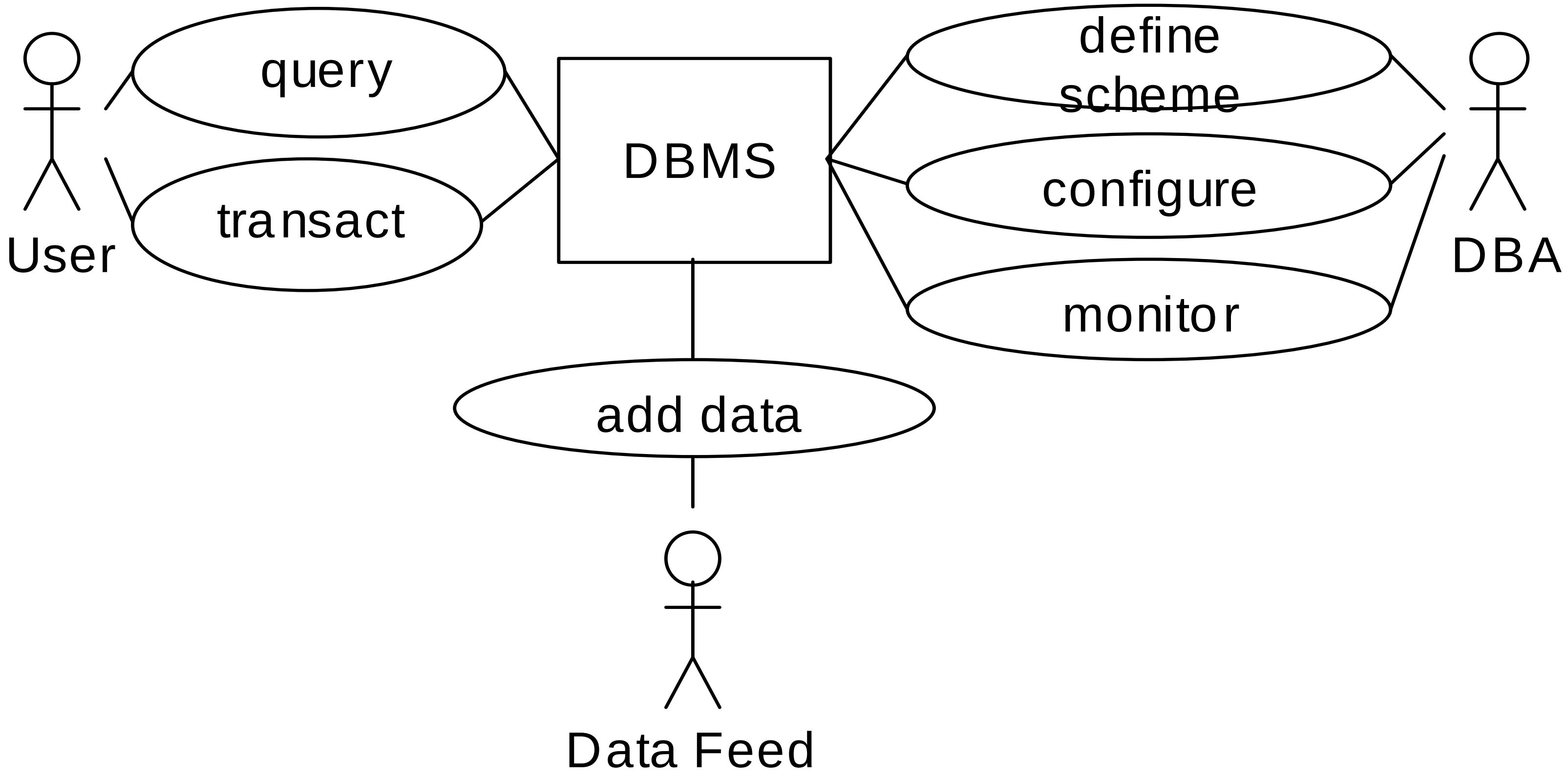
Abstract use cases model commonality.

Generalization/specialization models variability.

UML <<uses>> relationship can model delegation.

Abstract actors and generalization of actors models commonality and variability of users.

Example Use Case Model



Outline

- What is a Framework? What it is not!
- Framework Development, Application, Evolution.
- Development Concepts,Techniques and Models
- **Documentation for Application Developers**
- Wrap-up, Questions, Open Issues.

Documentation for Application Developers

- Overview of Types of Documentation

- Documentation for Application Developers
  - Framework Overview
  - Cookbook of Recipes
  - Graded Set of Example Applications

More … Chapter 21 in [Fayad 1999] & HotDraw documentation in Johnson, OOPSLA 1992

Documentation is Very Important

- Framework learning curve is too steep
  - typically 6-12 months
- Worse than general program understanding problem, because
  - design is very *abstract*, to factor out commonality
  - design is *incomplete*, needing extra classes to complete an application
  - design provides *too much flexibility*, not all this is needed by application at hand
  - collaborations and dependencies can be *indirect* and *obscure*

Documentation Approaches

- **source code** of framework
- source code of **example applications**
- **framework overview** stating domain and scope
- **cookbook** of **recipes** describing planned customizations in spiral, just-in-time fashion
- **reference manual** describing each class purpose, interface, *specialization interface*, constraints
- **design pattern** for each hotspot
- **behaviour specification** as *interface contract* or *interaction contract*
- **architecture** as a collection of *design patterns*

Recommended Documentation for Application Developers - 1

- Framework overview
  - 2-5 page description giving overview of domain, scope of framework, major concepts and collaborations, and planned customizations
  - recipe #1, used for selecting appropriate framework
- Cookbook of recipes
  - one recipe per kind and grade of customization
  - spiral from most common (easy) customizations to most infrequent (involved) customizations
  - just-in-time introduction of concepts and details only as needed to understand the recipe

Recommended Documentation for Application Developers - 2

- Cookbook of recipes (continued)
  - gradual introduction of advanced features for each customization
  - recipe is small number of easy steps, make clear how shared invariants are maintained
  - cross-reference to example applications
- Set of example applications
  - grade the set of examples from simple to advanced
  - synchronize with features needed for each recipe
  - separate simple customizations from advanced customizations of the same kind

Outline

- What is a Framework? What it is not!
- Framework Development, Application, Evolution.
- Development Concepts,Techniques and Models
- Documentation for Application Developers
- **Wrap-up, Questions, Open Issues.**

Wrap-up, Questions, Open Issues

- Summary
- Major Issues
- Audience Questions and Discussion
- Some Pointers to Literature

Summary - Development Issues

- General difficulty of design
  - decomposing problem, factoring classes, abstraction
  - commonality-variability analysis within application
- Plus
  - domain expertise, examples to drive analysis
  - commonality-variability analysis across applications
  - degree of bottom-up and top-down development
  - identify hotspots, i.e. where variability
  - identify required flexibility at hotspot
  - want narrow specialization interfaces
  - meta-level of abstractions, sometimes

Summary - Evolution Issues

- Lack of experience across maturity lifecycle
- How and when to transition to next stage of the lifecycle
- When to stop evolving a framework, and perhaps begin a new framework
- Refactoring tool support
- Choice of refactorings

- How quickly can we get to generators!

Open Issues

- Major Issues
  - learning curve
  - domain expertise required for framework developers
  - framework evolution
  - abstraction and meta-level abstraction

Questions, Anyone?

- Audience Questions and Discussion

References

M.E. Fayad, D.C. Schmidt, R.E. Johnson, “Building Application Frameworks”, Addison-Wesley, 1999.

Special Issue of CACM, October 1997.

I. Jacobson, M. Griss, P. Jonsson, “Software Reuse”, Addison-Wesley, 1997.

D. Roberts, R.E. Johnson, “Patterns for evolving frameworks”, Pattern languages of Program Design 3, Addison-Wesley, 1998.

K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, “FORM: A feature-oriented reuse method with domain-specific reference architectures”, Annals of SE, 5 (1998) 143-168.