Notes for OOPSLA '93

How to Design Frameworks

Ralph E. Johnson
University of Illinois at Urbana-Champaign

Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801

johnson@cs.uiuc.edu
(217) 244-0093

Frameworks

Interface design and functional factoring constitute the key intellectual content of software and is far more difficult to create or re-create than code.

Peter Deutsch

Design of an application or subsystem.

Set of abstract classes and way objects in those classes collaborate.

Use framework to build application by:

- Creating new subclasses
- Configuring objects together
- Modifying working examples

Frameworks

Framework prescribes how to decompose a problem.

Not just the classes, but the way instances of the classes collaborate.

- shared invariants that objects must maintain, and how they maintain them
- framework imposes a collaborative model that you must adapt to.

Inversion of control -- "don't call us, we'll call you".

Frameworks for user interfaces (MVC, MacApp,Interviews, ET++, etc.), operating systems, structured drawing editors, persistence, distributed objects, oscilliscopes, ...

Goal of Framework Design

- Build application from preexisting components.
- Use small number of types of components over and over.
- Write as little code as possible
  ultimate goal is zero code - build program by direct manipulation.

It shouldn't take a good programmer to build a good program.

How to Design Frameworks -- Copyright 1993 by Ralph E. Johnson

Framework
- is result of domain analysis
- describes how to decompose problem
- represented by a program

Reuse of
- analysis
- design
- code

Generalizations Require Concrete Cases

People think concretely, not abstractly.

Abstractions are found bottom-up, by examining concrete examples.

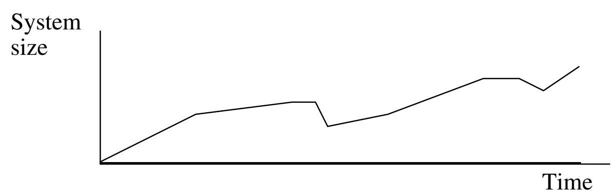
Generalization proceeds by
- finding things that are given different names but are really the same,
- parameterizing to eliminate differences,
- breaking large things into small things so that similar components can be found, and
- categorizing things that are similar.

How to Design Frameworks -- Copyright 1993 by Ralph E. Johnson

Example-driven Design

Examples $\xrightarrow{\text{Generalization}}$ Framework

Generalization is iterative, and lots of small changes are interspersed with a few major changes that represent breakthroughs in ways of looking at the problem.



To generalize faster:

- get different points of view
- explain/defend current design

Ideal Way to Develop Framework

Analysis → Design → Test

1) Analyze problem domain
   - Learn well-known abstractions.
   - Collect examples of programs to be built from framework. (Minimum of 4 or 5).
   - Deciding whether an example is legitimate depends partly on your vision for the framework.

2) Design abstraction that can be specialized to cover examples.
   - Develop theory to explain these examples.

3) Test framework by using it to solve the examples.
   - Each example is a separate application.
   - Performing a test means writing software.

Designing Abstractions

Design phase: look for commonalities, represent each idea once.

Good designers know many design patterns, techniques that they know tend to lead to good designs.

- implies that experience is needed

Rules of thumb indicate ways to change design to fix common problem.

Insight and ingenuity is always useful, but hard to schedule.

Why Ideal is Never Followed

Analyzing domain requires analyzing individual examples, and analyzing examples is already very hard.

- Only practical if examples have already been analyzed.
- Analyzing and implementing examples is large fraction of the cost of the project.

Typically some applications (the one you want to do next) are more important than others (the ones you have already done) so some examples are analyzed more than others.

Old applications work, so there is no financial incentive to convert them to use new software.

Good Way to Develop Framework

Pick two similar applications that need to be developed and that are obviously in your application domain.

Make sure your staff has some people who developed earlier applications in the same domain.

Divide project into three groups:

- framework group
  - Both gives and takes software
  - Considers how other applications would reuse framework
  - Develops documentation and training for framework
- 2 application groups
  - Try to reuse as much software as possible
  - Complains about how hard software is to use

Typical Way to Develop Framework

Realize that many applications are similar.

Develop next application in that domain in an OO language.

Try to divide software into reusable and nonreusable parts.

Develop second application reusing as much software as possible.

Surprise! Framework is not very reusable.

Fix it.

Develop third application reusing as much software as possible.

Surprise! Framework is not completely reusable.

Fix it.

...

Problems with Reuse as a Side-effect

Conflicting goals
- get system out on time
- make it reusable

Hard to pay for reusability

Hard to enforce reusability

Framework for Investing

Ward Cunningham
- portfolio -- collection of instruments
- instrument -- stock, bond, cash fund
  - has value, which can change daily
  - transactions affect it, and it causes transactions
- transactions
  - have a date
  - can affect several instruments at once

More on Ward's Framework

Value of instrument at particular date is a function of the transactions that have occured before that date.

Must be able to recalculate value of instrument at particular date when a transaction gets posted to an earlier date.

Solution:

- cache - keeps value at a particular date
- calculator - given a cache and a set of transactions, computes another cache
  - calculator can compute interest, taxes, etc. as a side effect of processing transactions.

Another Example

Accounting at a store

- journal -- collection of accounts
- account -- inventory, accounts payable (vender)
  - has value, which can change daily
  - transactions affect it, and it causes transactions
- transactions
  - have a date
  - can affect several accounts at once

Obvious Similarities

Investment system
- portfolio
- instrument
- transactions

Accounting at a store
- journal
- account
- transactions

Transactions are similar, and instrument and account are similar.

Each journal defines a new kind of account, and all accounts in a journal are the same.

The same kind of instruments appear in every portfolio, and a single portfolio has many kinds of instruments.

17

An Inventory Account

An invoice for a purchase increases inventory and increases amount owed to vender.

A sales transaction decreases inventory and increases amount sold to date.

Inventory #384		on hand	sold this month	this year	purchased this month
		30	100	1000	30
sales #6	(4)	26	104	1004	30
sales #8	(1)	25	105	1005	30
sales #13	(2)	23	107	1007	30
invoice #5	(50)	73	107	1007	80
sales #24	(1)	72	108	1008	80

18

Attributes

Each account has a set of *attributes*.

An attribute can be:
- total of a field taken from the transactions
- month-to-date, year-to-date, etc of another attribute
- function of several other attributes, i.e. inventory on hand is total purchased - total sold - shrinkage.

Each attribute has a name (SalesMonthToDate).

An attribute is time-dependent, i.e. reading an attribute always requires specifying a time.

Transactions

Transactions are just records: set of fields, some of which are multivalued.

Example: VendorInvoice

Vendor,

Date,

set of (Inventory type, Quantity, Price)

Payment due date, discount, etc.

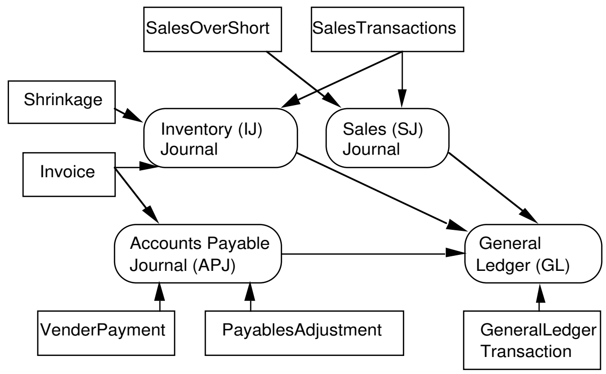
Purchase Order number,

Total (can be computed from other fields)

Constrained Transaction: contains a field that depends on the values of attributes of accounts

How to Design Frameworks -- Copyright 1993 by Ralph E. Johnson

Transactions and Attributes



Inventory account might have to read value from SalesTransaction, Invoice, and Shrinkage to compute an attribute.

Each kind of Transaction might give the value a different name.

Adding an attribute to an account requires specifying how to read values for the attribute from each transaction posting to the account.

How *Accounts* is a Framework

Abstract classes - Account, Transaction, Attribute

Way objects interact:

- Transaction -> Account - Transaction can post itself to set of Accounts
- Account -> Transaction - Account sorts Transactions that are posted to it
- Account -> Attribute - Account retrieves the value of an attribute at a particular time by delegating to the Attribute object
- Attribute -> Transaction - Attribute knows how to fetch value from Transaction

How to Design Frameworks -- Copyright 1993 by Ralph E. Johnson

How to use *Accounts*

Define Transactions and Accounts.

- define the Accounts to which a Transaction is posted

Define fields on Transactions.

Define Attributes on Accounts.

- define way Attribute is computed (+,*, mtd, ytd) by selecting appropriate subclass of Attribute
- for each Transaction posting to an Account, define how the Attribute gets a value

Examples

Good examples are crucial:

- for finding framework
- for teaching framework

<u>Examples define scope of the framework.</u>

Concrete examples → abstract framework

Finding Examples

Start with similar examples.
- should illustrate features that framework must support
- provides a base for generalization

Eventually examples should cover wide range.
- minimize number of examples

Look for examples that will break the framework.
- perhaps they will define limits of framework
- perhaps they will lead to generalization
- avoid lots of special cases

What Examples for *Accounts?*

We've worked most on:
- store
- investment

We have thought about:
- payroll
- personal finance
- banking
- budgeting

We should consider:
- insurance
- manufacturing inventory

Once *Accounts* can handle several small examples well, scale up.

Process of Framework Design

Framework is generalization of *implementation* of several problems.

- study solution domain as well as problem domain
- need to look at real programs
- need a wide range of carefully chosen examples
- generalization is the same process that occurs in all object-oriented design
  - find common abstractions
  - figure how to decompose problem into standard components
  - parameterize

There are common patterns of interaction that experts look for.

Example-driven: the examples you choose determine your framework

27

Scheduling Development

Software is not reusable until it has been reused.

Reusing software that is not reusable will point out reusability defects.

1) Domain analysis, collect examples	(small group)
2) Make preliminary design	(small group)
3) Implement examples	(many groups)

In addition to the framework design group, there probably needs to be a group for each example, unless the examples are small.

4) Fix framework and examples	(many unhappy groups)

As usual, every order of magnitude increase in the number of users will illustrate some (reusability) defects that you haven't detected before.

28

Design Patterns

Classes in framework can have many relationships between them.
Designers look for design patterns to increase reusability.

Superclass/subclass
Part/whole (component/container)
Builder/product
Delegator/delegatee
Double-dispatching

29

Composition vs. Inheritance

Two ways of reusing behavior
- inheritance
- composition

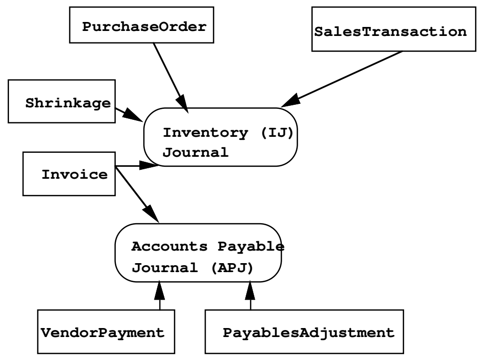
Inheritance
- easy to override inherited methods
- static (easier to reason about)

Composition
- less programming
- change behavior at run-time

One relationship can often be changed to the other.

30

Another Look at Accounts and Transactions



All Transactions posted to an Account must handle the same set of operations. Thus, all Transactions posted to an Account can be thought of as subclasses of a single subclass of Transaction.

SinceInvoice is posted to both InventoryAccounts and PayablesAccounts, it is a subclass of both InventoryTransaction and PayableTransaction.

Multiple inheritance!

Simulating M.I. with Composite Objects

Multiple inheritance can always be simulated by dividing an object into pieces.

<u>Sometimes</u> this improves a design.

Idea:

Suppose C is a subclass of S1, S2, ..., Sn. Give C components C1, C2, ..., Cn, where Ci is a subclass of Si for 0<i<n. Each Ci has a reference to C, which has a reference to all the Ci, so each component can access any other.

For *Accounts*:

Instead of making Invoice be a subclass of InventoryTransaction and PayablesTransaction, make it be a composite transaction with subtransactions that are InventoryTransactions and PayableTransactions.

Composites

A Composite for T is a subclass of T that combines a group of instances of subclasses of T and lets them act like a single instance of T.

Invoice is a composite Transaction because it combines a group of simpler Transactions and lets them act like a single instance of Transaction.

Can we find composite Accounts?

Journal, Portfolio are groups of Accounts. Does it make sense to consider them Accounts?

- what happens when transaction is posted to them?
- what does it mean to query their attributes?

33

CompositeAccount

What happens when Bob buys 100 shares of IBM stock?

The transaction "Bob buys 100 shares of IBM stock" is posted to Bob's Portfolio, which in turn posts a transaction to an IBM stock account that is part of the portfolio.

The value of Bob's portfolio is the sum of the values of the accounts in it.

What happens when an transaction is posted to a composite account?

- ask the transaction for the name of the component account, and then post the transaction to the component account

What does it mean to query an attribute of a composite account?

- query that attribute of the component accounts and return the sum

34

Refactoring to Help Find Composites

If an operation is implemented by performing a second operation on a component then make sure that the two operations have the same name.

read file -> read disk

draw picture -> draw elements of picture

For Accounts:

Post transaction to journal -> post transaction to account

Result: CompositeAccount is an account made up of other accounts.

Recursion Introduction

The way to find composites.

If message names cannot be identical, make them as similar as possible.

Often requires thinking of a more general name.

Often leads to new abstract superclass.

Classes with a collection of components become composites.

Strategy Object

A strategy is an algorithm represented as an object.

Representing an algorithm as an object means:

- easy to replace one algorithm with another
- can change dynamically
- can make a class hierarchy of algorithms
- can encapsulate private data of algorithm
- can define an algorithm in one place

Attributes as a Strategy

'Attribute' is responsible for computing the value of an attribute of an account at a particular point in time.

Different subclasses of Attribute can compute the value in different ways.

Representing an attribute as an object means:

- easy to change from total to MTD
- can make a class hierarchy of attributes, so it is easy for a programmer to make a new kind of attribute
- encapsulate indices into tables of intermediate values
- eliminates tag checking, and makes Account smaller

How to Design Frameworks -- Copyright 1993 by Ralph E. Johnson

Represent Common Features by Components

Each attribute of an Account:

reads a field from each transaction posted to the Account

applies a function (+) to list of values to produce a single value

if MTD or YTD then selects subset of transactions

Options:

1) Factor commonality into functions in Account, and have each function representing an attribute call them. Attributes are then simple and stylized, and so easy to implement.

2) Make attribute an object, which is responsible for extracting a field from transactions, for knowing which function to apply to values it extracts, and for knowing which transactions to extract values from.

Represent Common Features by Components

An alternative to multiple inheritance.

If some feature varies from class to class, but some classes have the feature in common, then represent the feature as a component. Make a class hierarchy to represent the variations in the feature.

How to carry out the refactoring:

Make a new class to represent the feature.

Add new component to existing classes.

Migrate functions and variables from existing classes to component.

Eliminate Case Analysis

Replace conditional tests of state with polymorphism

- eliminate flags
- encode state in class
- make a subclass for each case
- make an operation for each conditional statement
- eliminate case statements, if statements

Breaks class into smaller classes -- its subclasses.

Example:

Attribute -> MTDAttribute, YTDAttribute, TotalAttribute

41

Distributing Algorithm Across Objects and Subclasses

Advantages

- instead of modifying case statements, add a new subclass
- easier to parameterize
- can use inheritance to make new options

Disadvantages

- program is spread out,
  + harder to understand
  + harder to replace algorithm
- state of object can change, but class can not

  This means that changing an Attribute from MTD to YTD requires replacing it.

42

Designing Attribute Class Hierarchy

Attribute
- YTDAttribute
- MTDAttribute
- TotalAttribute

When objects do not come directly from the problem domain, it is often hard to design good abstract classes from first principles.

Usually best to design abstract class by generalizing from concrete subclasses.

Finding Abstract Classes/ Generalizing Concrete Classes

Create empty superclass, move common code/variables to it.

Problem: how to make common code.

- rename functions to give classes same interface
- decompose functions to factor out differences

Usually code in subclasses is almost, but not quite, the same. Must abstract out differences, and move the rest to superclass.

Result is that subclasses have more, but smaller, methods.

Decompose Functions

Factor out differences in two functions by:

- Find code sequences in two functions that are different.
- Turn each code sequence into a new function in the same class as the function containing the code sequence.
- Replace different code sequences with calls to equivalent functions.

45

Move Code From Caller into Callee

Law of Demeter says don't operate on contents of another object.

Instead, move code that operates on its contents into the object.

Example

x->foo->fee(a, b) => x->fee(a, b)

Define fee function in class of x that performs fee() on component.

Extreme case when record is converted into an object.

Rule: convert expressions that manipulate an object into operations on the object.

This is how to find operations for Attribute.

46

Interpreter / Little Languages

Need to define one attribute in terms of other attributes:

Average = Total / Count

OnHand = Purchase - Sold - Shrinkage

Idea:

1) make a class hierarchy that represents nodes in abstract syntax tree (/, -, +, attribute)

2) define a function *value(Date)* for each class

An AttributeNode knows an Account and Attribute on that Account, and *value(Date)* will read the Attribute on that Account for that date.

value(Date aDate) for a PlusNode will return the sum of the values for *aDate* for its two descendents.

3) define +, -, /, etc functions in common superclass to return PlusNode, DifferenceNode, RatioNode, etc.

Factory

What should a composite account do if it can't find a component with a given name?

- report an error -- the transaction is invalid

  Example: invoice has an illegal inventory number

- create a component account

  Example: buying some stock when you've never owned it before

CompositeAccount has a ComponentFactory that can create the right kind of component account to handle a transaction.

1) reject every transaction

2) look in large dictionary of possibilities and copy what it finds

(Example of the Prototype pattern.)

3) examine transaction and create instance of Account

Factory: object responsible for creating other objects.

Changing White-box to Black-box

Journal Account (with subaccounts) Portfolio Account	→	Account SimpleAccount CompositeAccount (parameterized by its component Accounts and the names it gives them)
Make new Account by defining attributes as functions in new Account class.	→	Make new Account by setting its list of Attribute objects.

Summary of Refactoring

Many reusability bugs can be fixed by a small set of transformations.

Lower level

- moving functions and variables to components
- moving functions and variables to superclass
- breaking functions into smaller functions
- renaming functions, classes, and variables

Higher level

- finding abstract superclasses
- breaking class into components or subclasses
  - reuse by inheritance to reuse by components
  - eliminating case analysis

Summary of Patterns

There are a set of higher-level design patterns that make systems more reusable and that are goals of refactorings.

Most design patterns rely on composition and polymorphism

- Composite
- Strategy
- Factory
- Interpreter

Some design patterns rely on inheritance

- Abstract classes with template methods

Design Patterns: Micro-Architectures for Reusable Object-Oriented Software
Erich Gamma, Richard Helms, Ralph Johnson, John Vlissides, in mid 1994.

Grand Summary

Developing reusable software is hard.

Reusable software developed iteratively

- generalize from concrete examples
- look for design patterns
- refactor to increase reusability