Secrets of Developing With Frameworks

Don Roberts

Dept. of Computer Science

1304 W. Springfield Ave.

Urbana, IL 61801

http://st-www.cs.uiuc.edu/~droberts

droberts@cs.uiuc.edu

First Rule of Framework Design

Don't. Buy one, instead.

Frameworks

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

Peter Deutsch

A Framework is:

- reusable design of an application or subsystem
- represented by a set of abstract classes and the 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".

Why We Are Interested

- Frameworks are the key to OO reuse
- Reusable objects need reusable context
- Frameworks reuse analysis and design, not just code.
- Frameworks reuse overall architecture, not just components.

Relevant Principles

- Frameworks are abstractions: people generalize from concrete examples
- Designing reusable code requires iteration
- Frameworks encode domain knowledge
- Customer of framework is application programmer

Generalize from 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.

Frameworks are Generalizations

Frameworks are normal OO programs generalized to fit many examples.

Common ideas become abstract classes.

Objects parameterized to make them reusable.

Ad-hoc systems broken down into smaller parts.

Use inheritance to organize class library.

Finding Abstract Classes

Abstract classes are discovered by generalizing from concrete classes.

To give two classes a common superclass:

- give them common interface
  - + rename operations so classes use same names
  - + reorder arguments, change types of arguments, etc.
  - + refactor (split or combine) operations
- if operations have same interface but different implementation, make them abstract
- if operations have same implementation, move to superclass

Frameworks Require Iteration

Reusable code requires many iterations.

Basic law of software engineering

If it hasn't been tested, it doesn't work.

Corollary: software that hasn't been reused is not reusable.

Frameworks Encode

Domain Knowledge

Frameworks solve a particular set of problems.

Not necessarily application-domain specific, but domain specific. (GUI, distribution, structured drawing editor, business transaction processing, workflow)

If you aren't a domain expert when you start to build a framework, you will be when you are done!

Customers are Programmers

Purpose of framework is to make it easier to build applications.

Apply these slogans to application programmers:

- The customer is always right.
- We are customer-driven.
- Understand your customer.

Example-driven Design

Examples —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

1) Analyze problem domain
   - Learn well-known abstractions.
   - Collect examples of programs to be built from framework. (Minimum of 4 or 5).

2) Design abstraction that covers 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
- *Design Patterns: Elements of Reusable Object-Oriented Software* Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley

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.

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 incentive to convert them to use framework.

Good Way to Develop Framework

Pick two similar applications.

Include developers experienced in the same domain.

Divide project into:

- framework group
  - Give and take software
  - Consider how other applications would reuse framework
  - Explain and teach framework
- 2 application groups
  - Try to reuse as much software as possible
  - Complain about how hard software is to use

Typical Way to Develop Framework

Notice that many applications are similar.

Develop next application in that domain in an OO language.

Divide software into reusable and nonreusable parts.

Develop next application reusing as much software as possible.

Surprise! Framework is not very 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

Another Strategy

Design framework - prototype several small applications.

Build real application.

Refactor and extend framework and old applications.

Build real application

Refactor and extend framework and old applications.

...

White-box vs. Black-box

White-box ⟵ ⟶ **Black-box**

Customize by subclassing

Emphasize inheritance

Must know internals

Simpler, easier to design

Harder to learn, requires more programming.

Customize by configuring

Emphasize polymorphism

Must know interfaces

Complex, harder to design

Easier to learn, requires less programming.

Design Patterns

Design patterns emphasize polymorphic composition over inheritance.

Show how to represent something that changes as an object

- Strategy -- algorithm
- Abstract Factory -- classes of products
- Mediator -- interaction between objects

Make designs more reusable and extensible

Application Generators

It is easy to draw pictures of an application made from a black-box framework.

It is easy to generate code from a picture for a black-box framework.

It is easy to make special-purpose visual programming languages for black-box frameworks.

Visual programming languages let non-programmers build applications.

Summary: Patterns for Developing Frameworks

1) Three Examples
2) White-box Framework
3) Component Library
   - Build applications and add components to library
4) Hot Spots
   - Separate Changeable from Stable Code
   - Design Patterns

Summary: Patterns for Developing Frameworks

5) Pluggable Objects

6) Fine-grained Objects

7) Black-box Framework

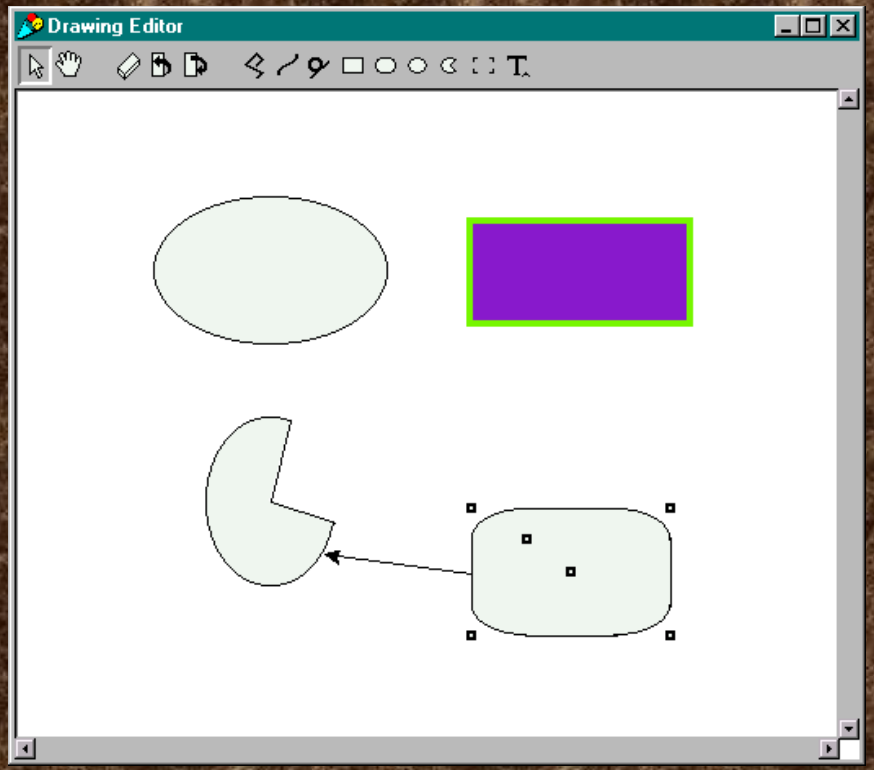
8) Visual Builder

9) Language Tools

http://st-www.cs.uiuc.edu/users/droberts/evolve.html

HotDraw

Drawing editor framework for Smalltalk designed by Ward Cunningham and Kent Beck at Tektronix. John Brant has evolved it extensively since then.



Hotdraw Ex1: Drawing Editor

Drawing - the “surface”

Figures - The different shapes that can be constructed

Tool - The different mouse cursors that manipulate and create the figures.

Handles - Boxes that appear when the figure is selected

Hotdraw Ex.2 - PERT Chart

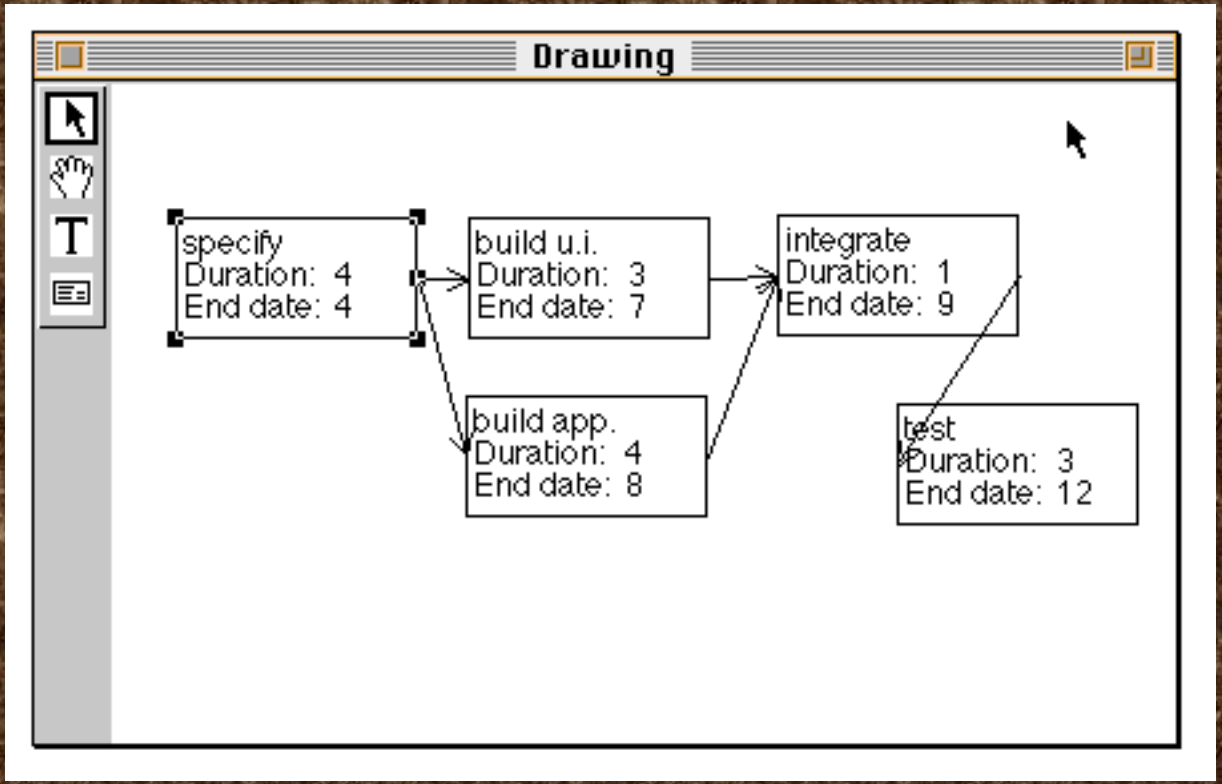
New classes:

PERTEvent: figure with a duration and a completion time.

- subclass of CompositeFigure
- add handle to connect events
- connection causes constraint between completion time of one PERTEvent and starting time of the other
- internal constraints between starting time, duration, and completion time of a PERTEvent

PERTChartEditor: subclass of DrawingEditor that specifies the palette.

PERT Chart



Figure

A figure knows its bounding rectangle on the screen.

bounds

A figure can display itself.

displayFigureOn:

A figure knows the drawings and other figures that depend on it and notifies them when it changes.

A figure can create handles for itself.

handles

Drawing

Drawing keeps track of figures

add:, add:behind:, remove:, bringToFront:, sendToBack:

Drawing can be displayed

displayOn:

Drawing can be animated (default does nothing)

step

Tool (original)

Tool has an icon (for pallette) and a cursor (for canvas).

Tool acts like a part of the controller. The drawing controller delegates all operations to the current tool, so changing the current tool changes the behavior of the editor.

Most Tools only define a few operations.

Operations performed by controller

activate, deactivate

backspace, controlCharacter:, type:

press, pressBackground, pressFigure:, release

Handle (original)

A handle is a little black box that is attached to a figure and that does something to the figure when you press it.

One operation -- invoke: aView

-- perform special function

-- default is to follow cursor

Domain Knowledge Reuse

HotDraw automatically reuses good graphics techniques.

Damage: Drawing keeps track of area that is *invalid* or *damaged*. Figures damage the smallest area that they know they have changed. Drawing minimizes work in redrawing damaged area.

Double buffering: Program seems faster when drawing is done on a temporary bitmap and moved to the screen all at once.

Problem with Old HotDraw

Tools are hard to build

- have to decide which methods to redefine
- usually duplicate code
  - code for following mouse
  - code in other tools

Change Tool to be compositional

- icons for pallette and cursor
- table mapping events to Commands

Commands

Command knows its figure and its tool

effect - evaluate Command

MouseCommands

initialize: aPoint

gets called at start of gesture

moveTo: currentPoint

gets called repeatedly throughout gesture as mouse is moved

TextCommands

initialize

keyboardEvent: anEvent

Tool doesn't have to be subclassed, just a table of Commands.

	Figure	Drawing	TextFigure	CmdHndl	CnstHndle
LineFigureTool	Comm				
SelectionTool	Comm	Comm		Comm	Comm
TextTool	Comm	Comm	Comm		

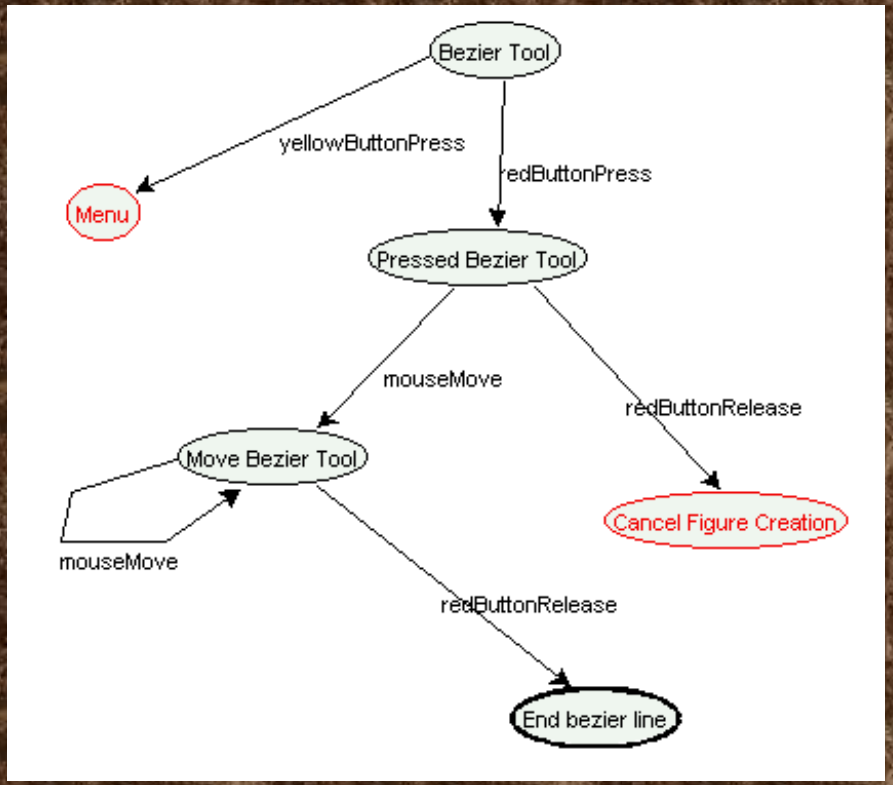
Table defines commands for combinations of Tool and Figure class

Handle is now just a Figure with an embedded Command.

Events

Older Smalltalks were polling based. Newer versions are event-driven.

Tools are now state machines that transition on events and figures.



State Pattern

Tools are now implemented with the state pattern combined with the Command pattern. The command pattern is implemented simply with Smalltalk blocks.

Tool - holds onto the current ToolState. Never subclassed.

ToolState - holds onto the command executed when this state is entered. Never subclassed.

Result: New Tools can be built without adding any classes.

Advantages of Black-box Frameworks

Replace reuse by inheritance with reuse by object composition

→ make programs without defining new classes.

Ultimate black-box framework lets you plug objects together with visual language.

Application development doesn't require programming.

Disadvantages of Black-box Framework

Black-box frameworks harder to design and harder to make powerful.

Black-box framework tends to have

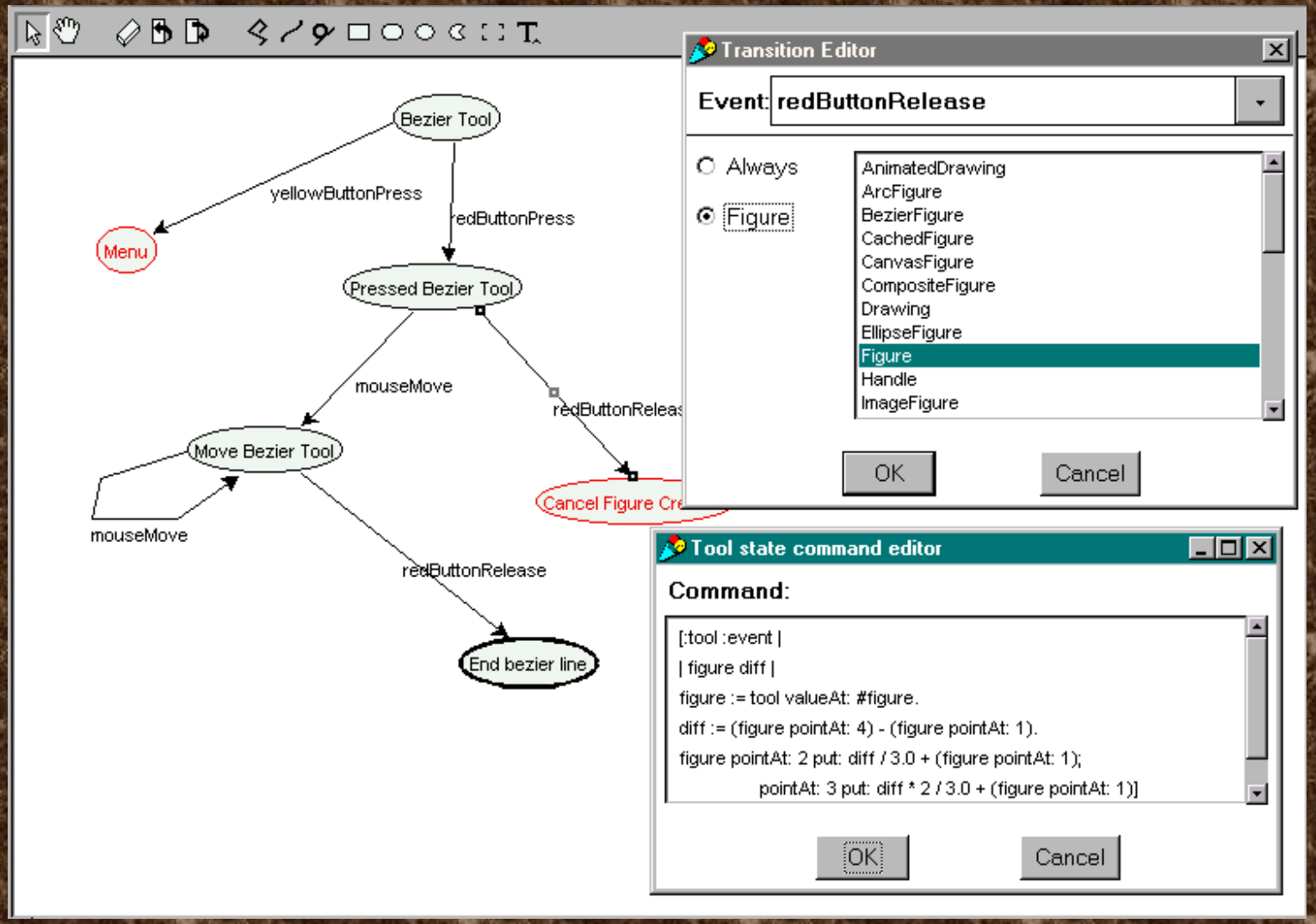
- more kinds of objects
- more artificial kinds of objects
- more complex relationships between objects
- more objects

Not-quite-ultimate framework forces you to debug more complex system.

Code for Bezier Tool

```
Tool states at: 'End bezier line' put: (EndToolState name: 'End bezier line' command:
    [:tool :event | | figure diff |
        figure := tool valueAt: #figure.
        diff := (figure pointAt: 4) - (figure pointAt: 1).
        figure pointAt: 2 put: diff / 3.0 + (figure pointAt: 1); pointAt: 3 put: diff * 2 / 3.0 + (figure pointAt: 1)]).
Tool states at: 'Bezier Tool' put: (ToolState name: 'Bezier Tool' command: [:tool :event | tool cursor: Cursor crossHair]).
Tool states at: 'Pressed Bezier Tool' put: (ToolState name: 'Pressed Bezier Tool' command:
    [:tool :event | | figure point |
        point := tool cursorPointFor: event.
        figure := BezierFigure start: point stop: point.
        tool valueAt: #figure put: figure.
        tool drawing add: figure]).
Tool states at: 'Move Bezier Tool' put: (ToolState name: 'Move Bezier Tool' command:
    [:tool :event | (tool valueAt: #figure) stopPoint: (tool cursorPointFor: event)]).
(Tool stateFor: 'Move Bezier Tool') redButtonRelease: ((SimpleTransitionTable new) goto: (Tool stateFor: 'End bezier line');
    yourself).
Tool stateFor: 'Move Bezier Tool') mouseMove: ((SimpleTransitionTable new) goto: (Tool stateFor: 'Move Bezier Tool'); yourself).
(Tool stateFor: 'Bezier Tool') redButtonPress: ((SimpleTransitionTable new) goto: (Tool stateFor: 'Pressed Bezier Tool'); yourself).
(Tool stateFor: 'Bezier Tool') yellowButtonPress: ((SimpleTransitionTable new) goto: (Tool stateFor: 'Menu'); yourself).
(Tool stateFor: 'Pressed Bezier Tool') redButtonRelease: ((SimpleTransitionTable new) goto: (Tool stateFor: 'Cancel Figure
    Creation'); yourself).
(Tool stateFor: 'Pressed Bezier Tool') mouseMove: ((SimpleTransitionTable new) goto: (Tool stateFor: 'Move Bezier Tool');
    yourself).
```

Visual Builder



Patterns and Refactoring

Refactoring

- changing the structure of a program, but not its function.
- the most common way to fix reusability problems.
- making a "hot spot" flexible
- often is applying a pattern
- tools can help

The Refactoring Browser

Reimplementation of the browser that includes many common refactorings.

- rename classes, variables, methods
- extract portions of methods into separate methods
- push methods, variables up and down the hierarchy
- move methods into other classes through references

Each refactoring is safe and fast.

Hints for Framework Design

Use object composition instead of inheritance

Incrementally apply patterns / lazy generalization

Framework should work out of the box

Size Matters

Small — Large

When you spot an abstraction that looks useful, generalize.

Periodically take time to clean up your design.

Explain design to your peers when they need to understand it.

Must decide which frameworks to build.

Must schedule new releases.

Documentation.

Training

Scheduling Development on a Large Framework

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)
4) Fix framework and examples (many unhappy groups)

Strategic Concerns

Developing a framework is expensive, so look before you leap.

- Framework development requires long term commitment.
- Pick frameworks that will give you competitive advantage.
- Start small and work up.
  - get experience with OOP
  - select and train good abstractors
  - build small frameworks first
  - generalize existing systems

Customers are Crucial

Get customers involved early, and use their feedback.

Make your initial customers succeed.

First set of customers are really part of the development team.

- their input is critical
- they will suffer from redesign of framework
- they must feel that their contribution to framework is important

Reuse Scenarios

Ideal: Successive versions of your framework meet the needs of a growing customer base.

Actual: Projects may customize the initial framework, and even start competing streams of development.

Dealing with Iteration

Don't claim framework is useful until your customers say it is. This will take two or three versions.

Keep customer base small while framework is evolving.

A successful framework is a living framework, evolve it to meet new customer needs.

Don't constantly tinker. Plan releases and coordinate with customers.

Documentation and Training

Documentation for framework costs several times usual

- how to use
- how to extend / how it works

Software must be understandable to be usable.

Improving documentation can make software more reusable.

Documentation and Training

Base documentation on examples.

Must debug documentation and training.

Documenting system shows how to change it.

Framework developers must be intimately involved.

NIH vs. TILI

Problem with reuse is NOT fault of customer.

It is hard to make software reusable.

- Must be abstract and powerful - theory of application domain.
- Must be customizable - theory of what users want to change.
- Must be easy to understand
  -- simplicity is crucial
  -- needs good documentation

More Information

You can find papers on frameworks, on patterns, and on the example I used (HotDraw) from Ralph's home page.

http://st-www.cs.uiuc.edu/users/johnson/