Reuse Contracts as a basis for investigating reusability of Smalltalk code

Koen De Hondt
Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel

kdehondt@vub.ac.be http:/progwww.vub.ac.be/

Overview

- **Problems with reuse**
- Problems with evolution
- What are reuse contracts?
- Reuse contracts at work
- Examining class hierarchies based on reuse contracts
- Reuse contract research
- Exercises: introduction to the browser

How do You Reuse a Class?

- Cloning (copy and paste)
- Inheritance / method overriding
- Composition / delegation

Reuse by Cloning

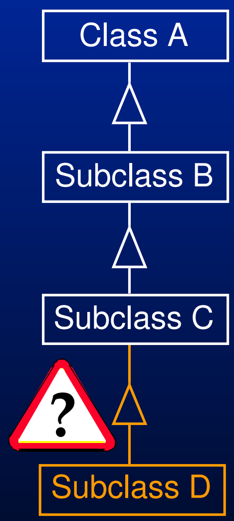
- Reused “components” are not easily adaptable
  - no support is provided for adaptation/reuse
- No relation between original and result
  - difficult to maintain since bug fixes and upgrades are not propagated to the derived application (proliferation of versions)

This kind of reuse should be avoided

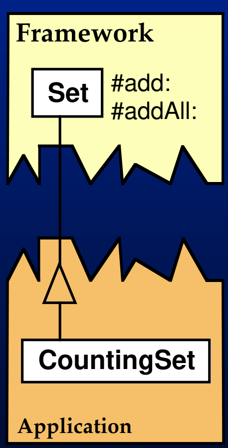
Reuse by Inheritance

How do you determine

- what to reuse (inherit)?
- what to adapt (override)?
- what to write from scratch?



Example: Make a Subclass of Set



What to override?

- #add: if #addAll: uses #add:
- #add <u>&</u> addAll: if #addAll: does not use #add:

A CountingSet is a Set that counts all added elements

Reuse by Composition

How do you determine

- what to reuse (what to compose, what to delegate)?
- what to adapt (how to compose)?
- what to write from scratch?

Reusing a Class is Hard

- Current OOA/OOD notations do not provide enough information to reuse a class
- Usually, developers do not document how a class can be reused, they only document what each method does
- If a class comment contains reuse information, it usually has the form of a cookbook

Reusers are compelled to inspect the source code

Inspecting the Source Code

- To reuse a class:
  - inspect the class
  - inspect all its superclasses
  - inspect all the classes it co-operates with
- Source code inspection is error-prone
- If source code inspection doesn't work: talk to the developer (i.e. the expert)!

What are You Looking for?

- **Self sends**
- Super sends
- Abstract methods
- Template methods
- Default methods
- Methods that are overridden frequently
- Methods that are part of a design pattern
- **Co-operation with other objects/classes**
- ...

Reusers need the specialisation interface

Self Sends are Important

- Self sends & template methods & abstract methods reify the design of a class
- Method decomposition
  - distinguish “core” methods from “peripheral” methods
- Using self sends = planning for reuse
  - fine-grained overriding of methods

Self Sends: Planning for Reuse

ApplicationModel in VisualWorks <u>2.5</u>

```
openInterface: aSymbol
    builder := self builderClass new.
    ...
    “a lot of expressions here”
    ...
```

can be reused with other builders

same external interface
(#builderClass is private)

ApplicationModel in VisualWorks <u>2.0</u>

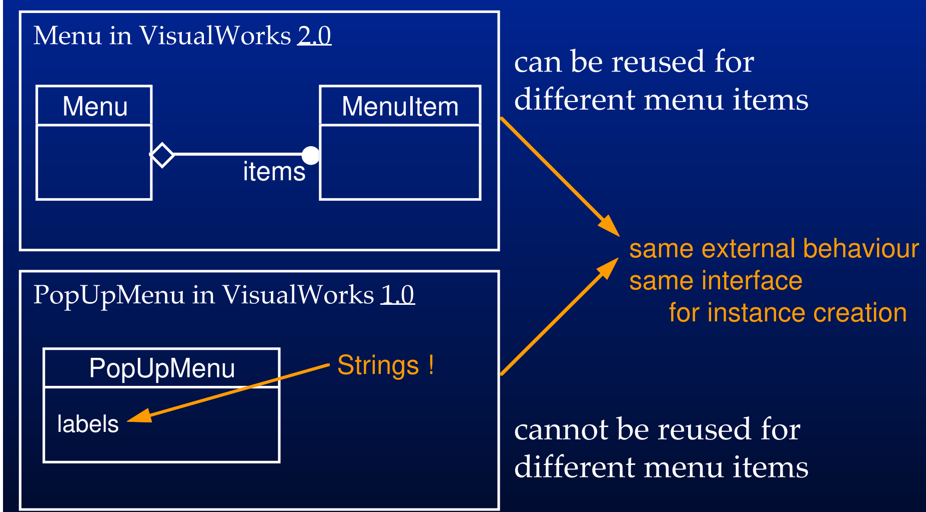
```
openInterface: aSymbol
    builder := UIBuilder new.
    ...
    “a lot of expressions here”
    ...
```

cannot be reused with other builders without overriding **<u>all</u>** methods that refer to UIBuilder

Co-operation with Other Objects/Classes is Important

- Delegation of responsibilities principle
- Using delegation= planning for reuse
  - a system can easily be extended by adding new classes
  - objects with “the same interface” can be substituted for each other

Delegation: Planning for Reuse



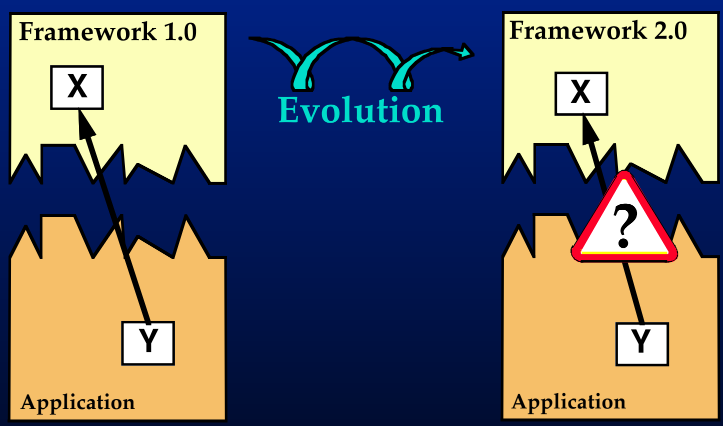
Overview

- Problems with reuse
- **Problems with evolution**
- What are reuse contracts?
- Reuse contracts at work
- Examining class hierarchies based on reuse contracts
- Reuse contract research
- Exercises: introduction to the browser

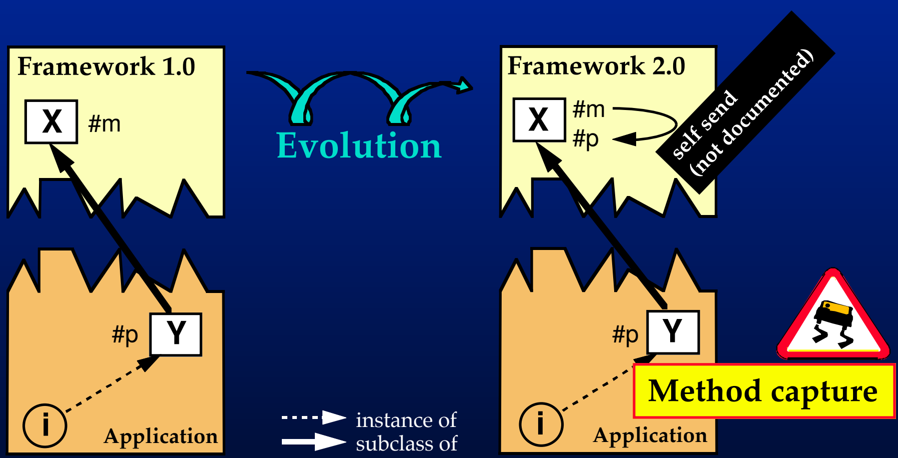
Evolution is Important

- Iterative development
  - a framework is never finished
- Changing requirements
  - functional: user requirements
  - non-functional: maintainability, adaptibility, reusability, customisability, ...

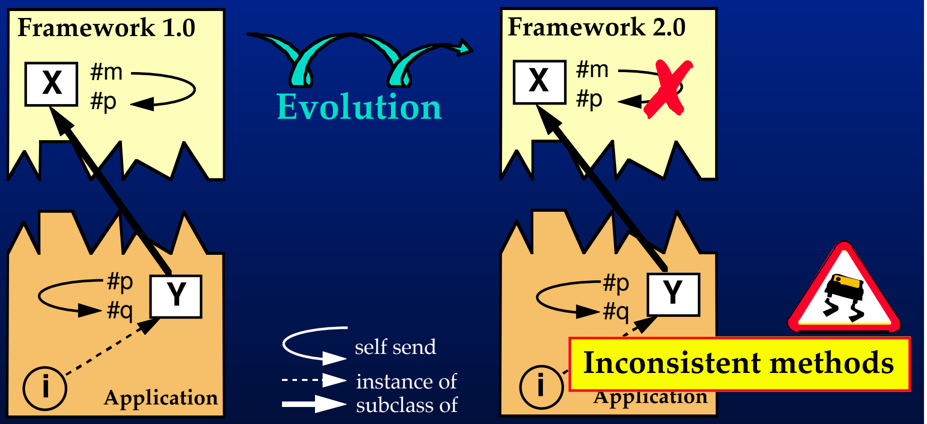
What to do When the Framework Changes?



Example Evolution Conflict (1)



Example Evolution Conflict (2)



More Evolution Conflicts

- Interface conflicts
  - the name of a reused method/class has been changed
  - a method that was added by a reuser has been introduced by the new version of the framework
- Unanticipated recursion
  - a method invokes another one in the application while the new version of the framework introduces an invocation of the first by the latter

Spotting Evolution Problems

- Unless the changes to the framework are well-documented (informally), the application developer is condemned to perform code inspection to determine what has changed
- Often evolution conflicts are not spotted until the application is running based on the new version of the framework

What are the Challenges?

- Supporting reuse
  - what can be reused, what must be adapted, and what must be built from scratch ?
  - formal documentation on how classes are reused
- Supporting evolution
  - change propagation
- Support for estimates/testing/metrics
  - feasibility of reusing a class
  - the cost of “upgrading” the class repository

Overview

- Problems with reuse
- Problems with evolution
- **What are reuse contracts?**
- Reuse contracts at work
- Examining class hierarchies based on reuse contracts
- Reuse contract research
- Exercises: introduction to the browser

Reuse Contracts

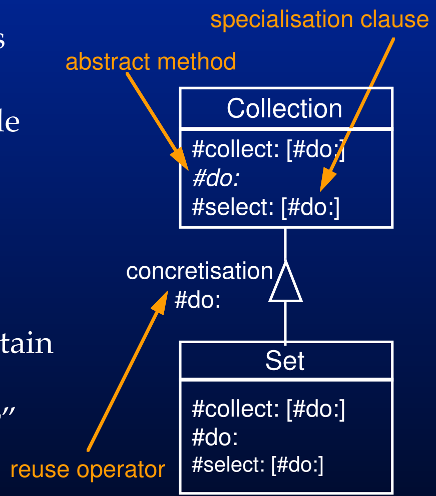
- Are contracts between the framework developer and the application developer
- State what assumptions can be made about reusable components
- State how components are actually reused

Reuse Contract Notation

- Notation based on OMT (UML)
- Methods are annotated with specialisation clauses to make the specialisation interface explicit
- "Reuse operators", or "modifiers", lay down how reuse is achieved

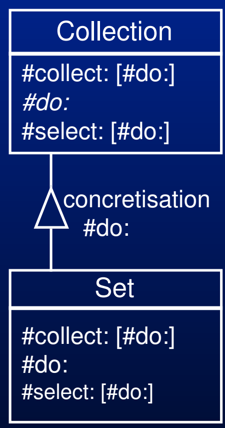
Reuse Contracts for Inheritance

- Enhance the interface of a class with specialisation clauses
- Identify what changes are made when a class is subclassed:
  - concretisation / abstraction
  - extension / cancellation
  - refinement / coarsening
- Specialisation clauses may contain names of methods invoked through self sends, and "super"



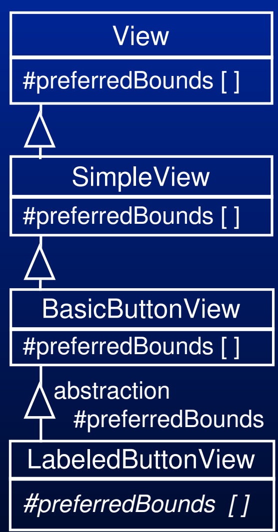
Reuse Operator: Concretisation

- Makes abstract methods concrete
- Does not change the specialisation clause of the concretised methods
- Design preserving
- Inverse = abstraction



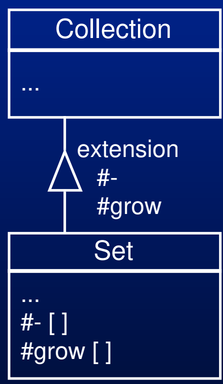
Reuse Operator: Abstraction

- Makes a concrete method abstract
- Design breaching
- Inverse = concretisation



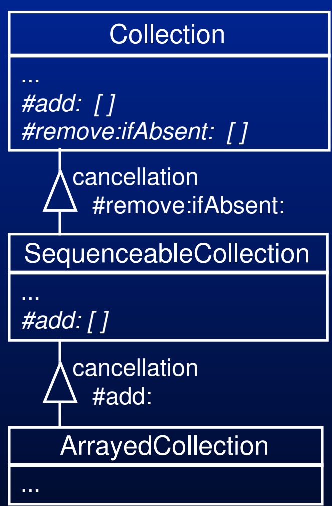
Reuse Operator: Extension

- Typically performed by an application developer to add application specific behaviour
- Adds new methods to the interface of a class
- Design preserving
- Inverse = cancellation



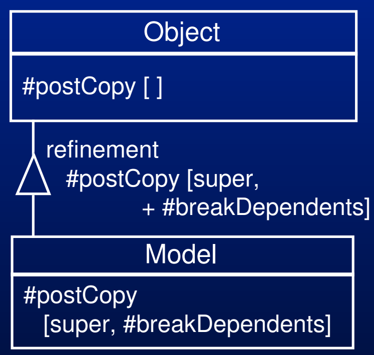
Reuse Operator: Cancellation

- Typically performed by an application developer to remove behaviour
- Removes methods from the interface of a class
- Design breaching
- Inverse = extension



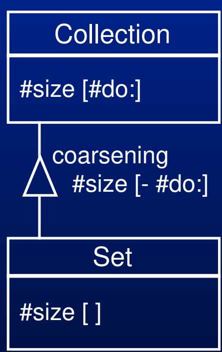
Reuse Operator: Refinement

- Adds elements to the specialisation clause of a method
- Used to e.g. :
  - reduce redundancy
  - specialise the behaviour of an existing method with an existing behaviour
- Design preserving
- Inverse = coarsening



Reuse Operator: Coarsening

- Removes elements from the specialisation clause of a method
- Used to e.g.:
  - optimize performance
- Design breaching
- Inverse = refinement



Reuse Operators

- Make a distinction between different kinds of inheritance
- State how a class is derived from its superclass
- Are orthogonal basic operators
- Usually, one subclassing step is a combination of several reuse operators

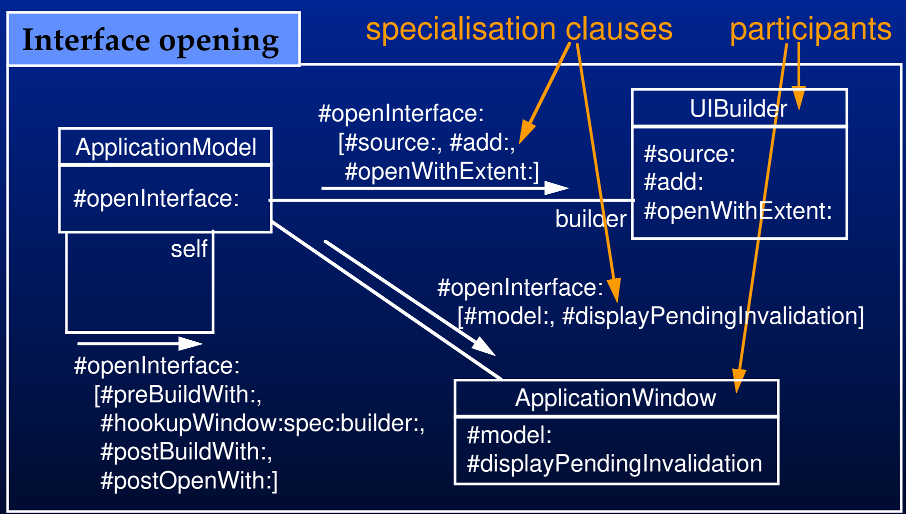
Frequently Used Combinations of Reuse Operators

- Extension & refinement
- Coarsening & cancellation
- Concretisation & refinement
- Concretisation & extension & refinement
- Coarsening & refinement = redefinition
- Coarsening & extension & refinement = factorization

Multi-Class Reuse Contracts (in short)

- Co-operating classes are put in one reuse contract; these classes are called “participants”
- Interfaces of classes as in reuse contracts for inheritance
- Specialisation clauses are extended with names of methods invoked on other classes
- Reuse operators identify what changes are made to a whole contract

Reuse Contract Notation



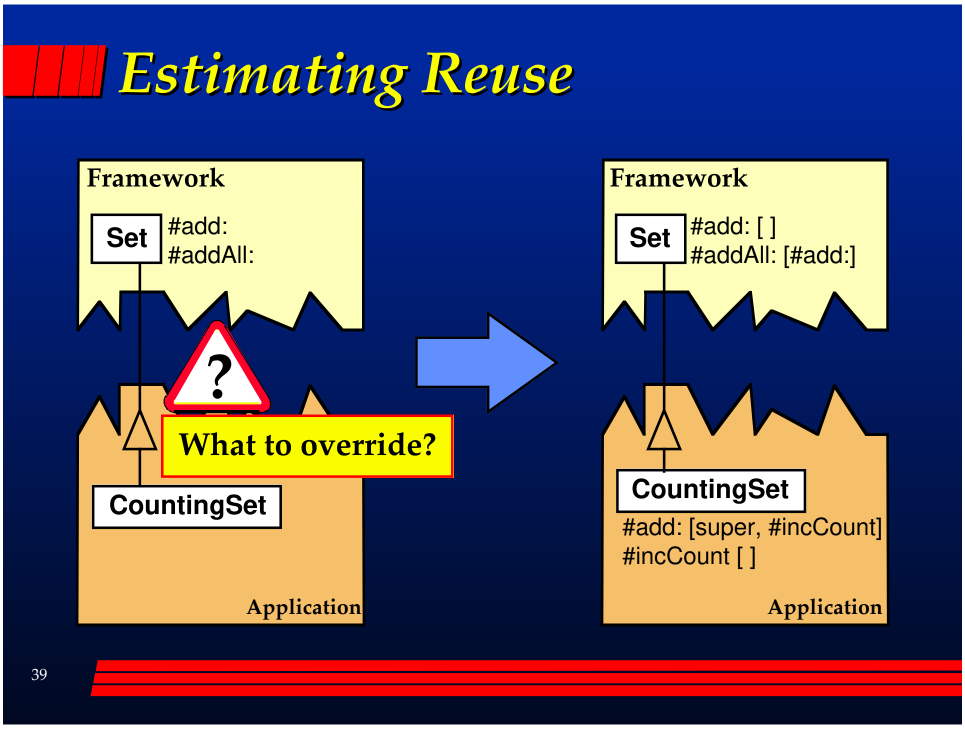
Overview

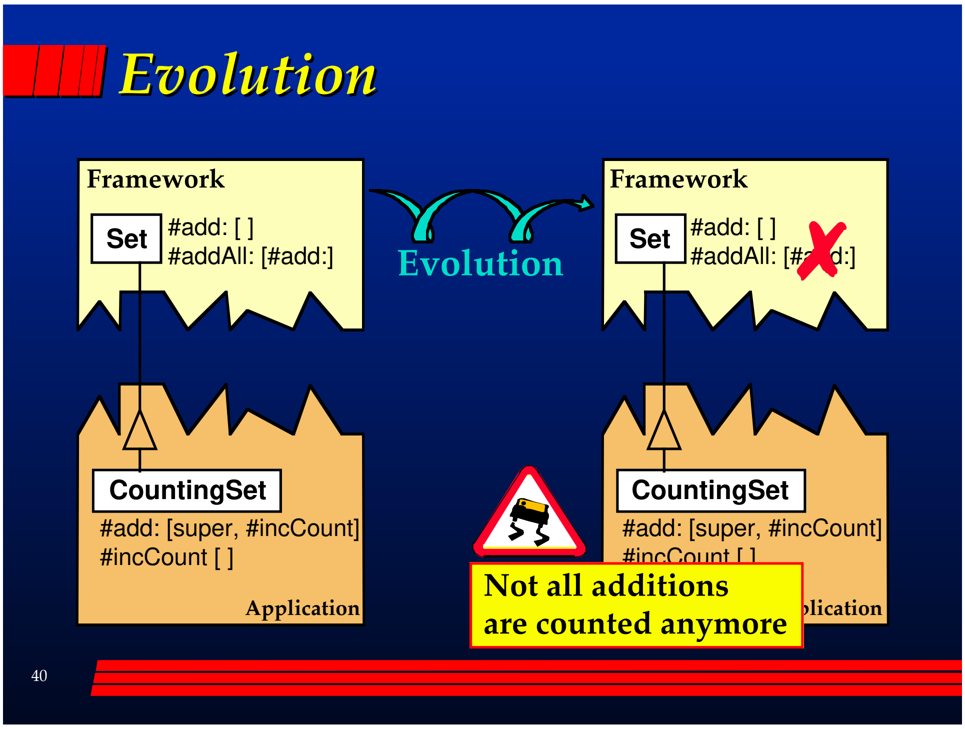
- Problems with reuse
- Problems with evolution
- What are reuse contracts?
- **Reuse contracts at work**
- Examining class hierarchies based on reuse contracts
- Reuse contract research
- Exercises: introduction to the browser

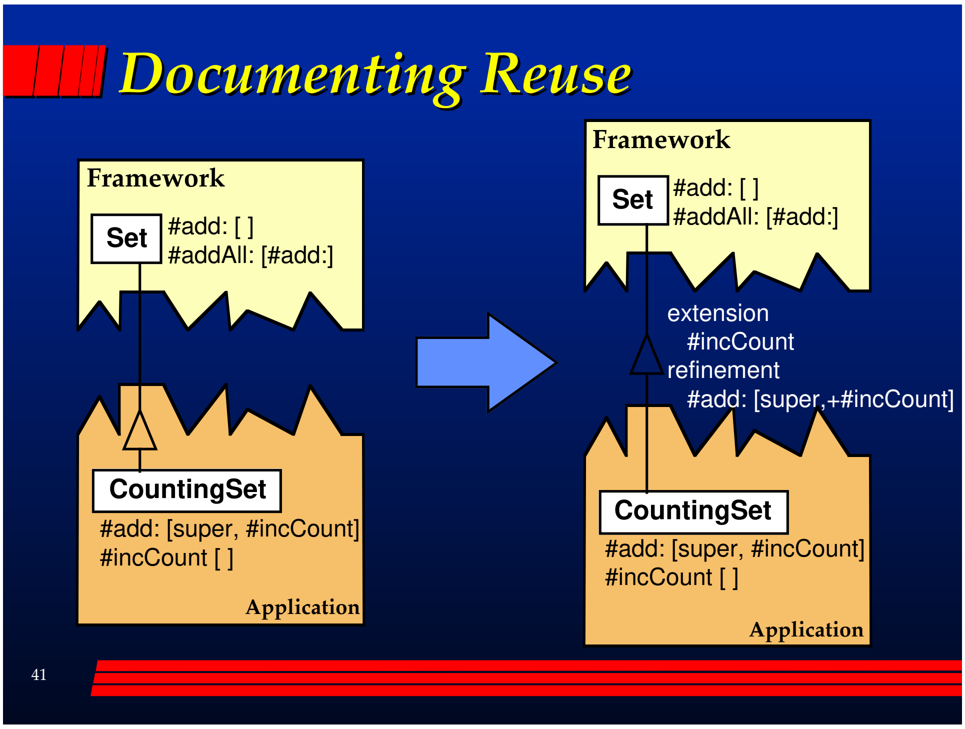
Reuse Contracts at Work

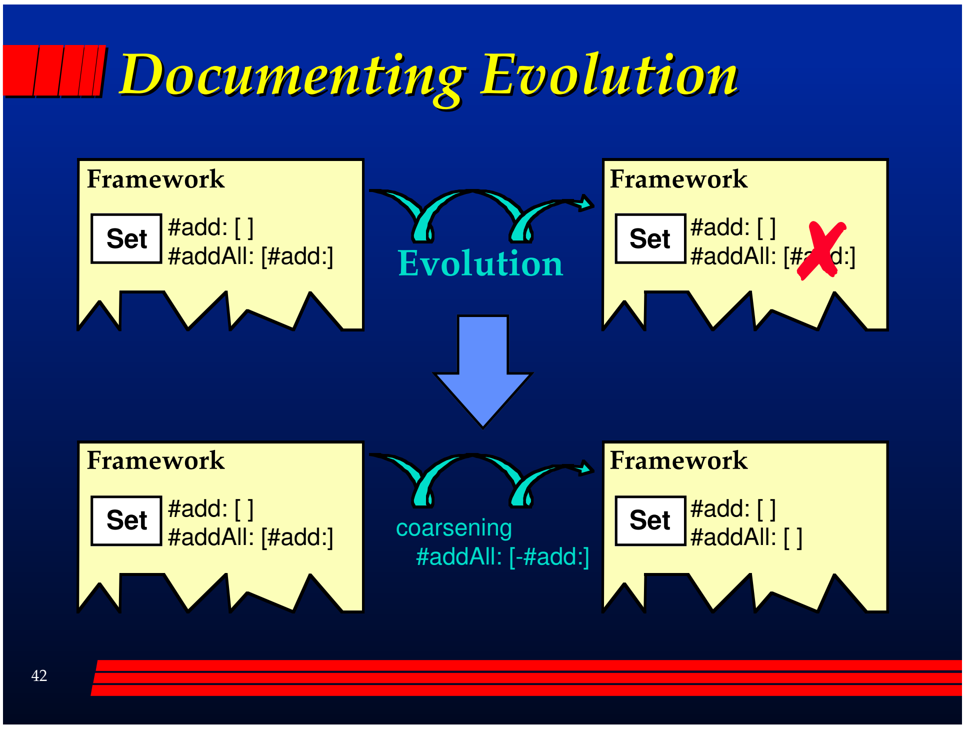
The formal nature of reuse contracts enables their use in a development environment

- code generation from reuse contracts
- impact analysis when a framework changes (assessing evolution conflicts)
- effort estimation for framework customisation
- extraction from source code

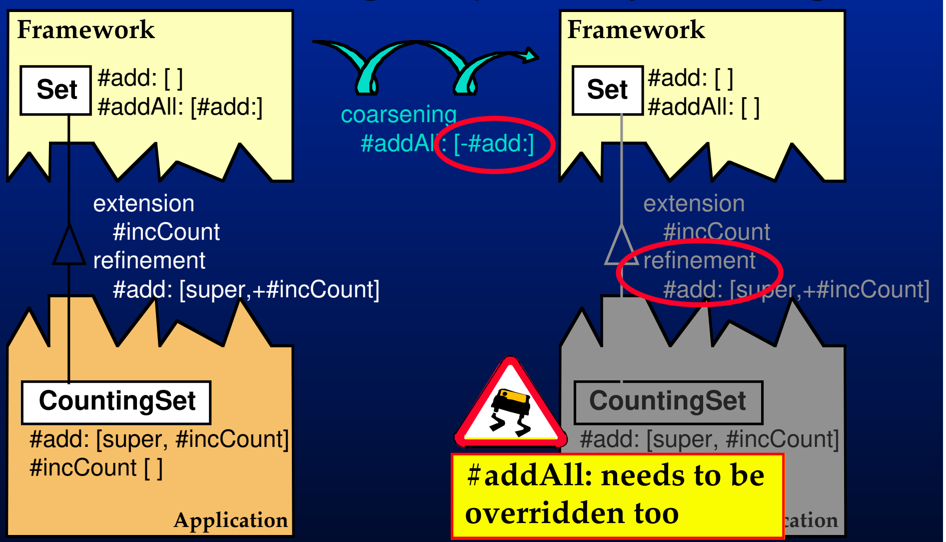

Estimating Reuse
Framework
Set
#add:
#addAll:
?
What to override?
CountingSet
Application
Framework
Set
#add: [ ]
#addAll: [#add:]
CountingSet
#add: [super, #incCount]
#incCount [ ]
Application
39


Evolution
Framework
Set
#add: [ ]
#addAll: [#add:]
CountingSet
#add: [super, #incCount]
#incCount [ ]
Application
Evolution
Framework
Set
#add: [ ]
CountingSet
#add: [super, #incCount]
Not all additions
are counted anymore
40


Documenting Reuse
Framework
Set
#add: [ ]
#addAll: [#add:]
CountingSet
#add: [super, #incCount]
#incCount [ ]
Application
Framework
Set
#add: [ ]
#addAll: [#add:]
extension
#incCount
refinement
#add: [super,+#incCount]
CountingSet
#add: [super, #incCount]
#incCount [ ]
Application
41


Documenting Evolution
Framework
Set
#add: [ ]
#addAll: [#add:]
Evolution
Framework
Set
#add: [ ]
#addAll: [#add:]
Framework
Set
#add: [ ]
#addAll: [#add:]
coarsening
#addAll: [-#add:]
Framework
Set
#add: [ ]
#addAll: [ ]
42

Estimating Impact of Changes

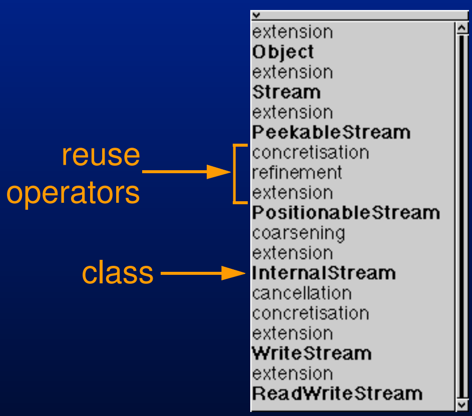


Overview

- Problems with reuse
- Problems with evolution
- What are reuse contracts?
- Reuse contracts at work
- **Examining class hierarchies based on reuse contracts**
- Reuse contract research
- Exercises: introduction to the browser

Extraction of Reuse Contracts

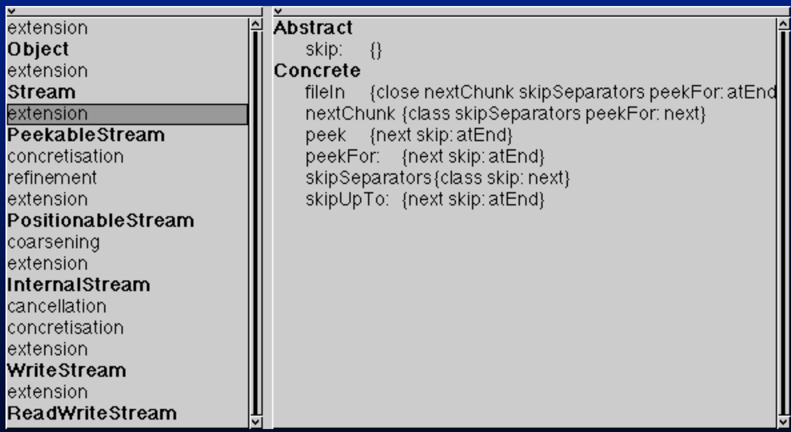
- Reuse contracts for inheritance can be extracted from Smalltalk code
- Each subclassing step is decomposed in a combination of maximum 6 different reuse operators



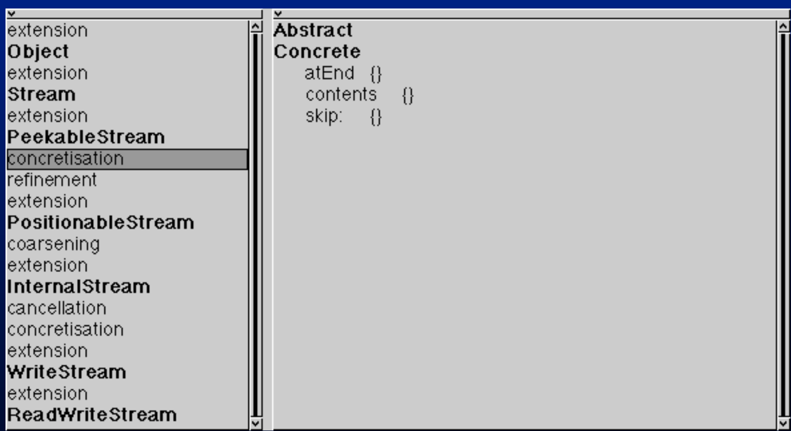
Too Much Extracted Information

- The extractor does not know which methods are important
- Interaction with a developer is required to strip implementation details

Inspecting Extracted Extensions

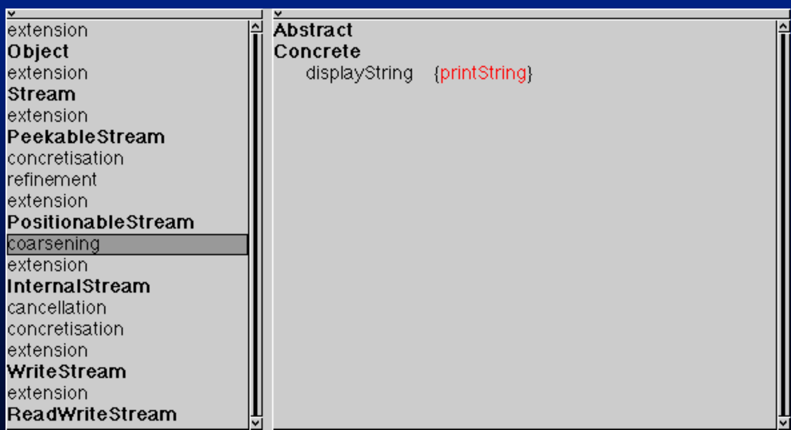


Inspecting Extracted Concretisations

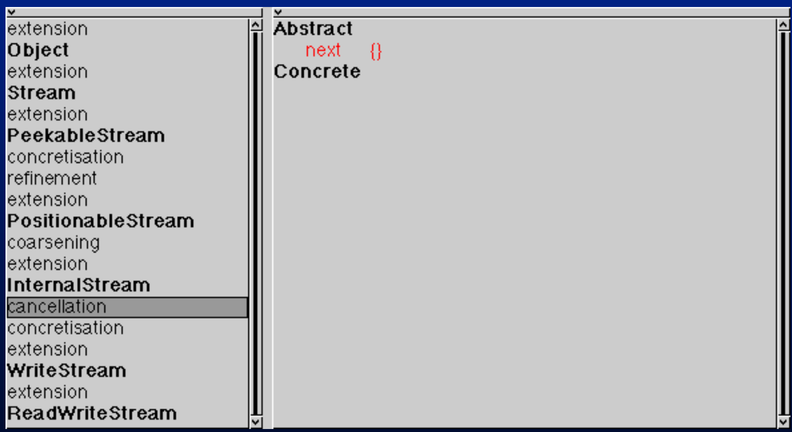


Inspecting Extracted Refinements

Inspecting Extracted Coarsenings



Inspecting Extracted Cancellations

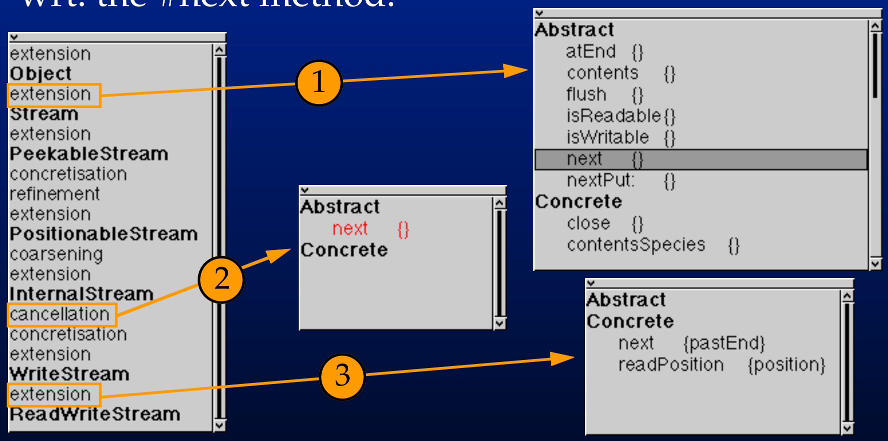


Spotting Bad Designs in a Class Hierarchy

- Look for design breaching reuse operators
  - they indicate methods that do not respect the design as laid down by a superclass
- Examine what happens with the affected methods in reuse operators that are applied later on

Spotting Bad Designs: Example

The Stream hierarchy is awkward
wrt. the #next method.



Impact of Bad Coding Style

- Bad coding style is troublesome for the extractor
  - e.g. only super send, bad super send, ...
- This has driven us to make qualitative assessment of source code possible
- An analysis tool is integrated in our browser

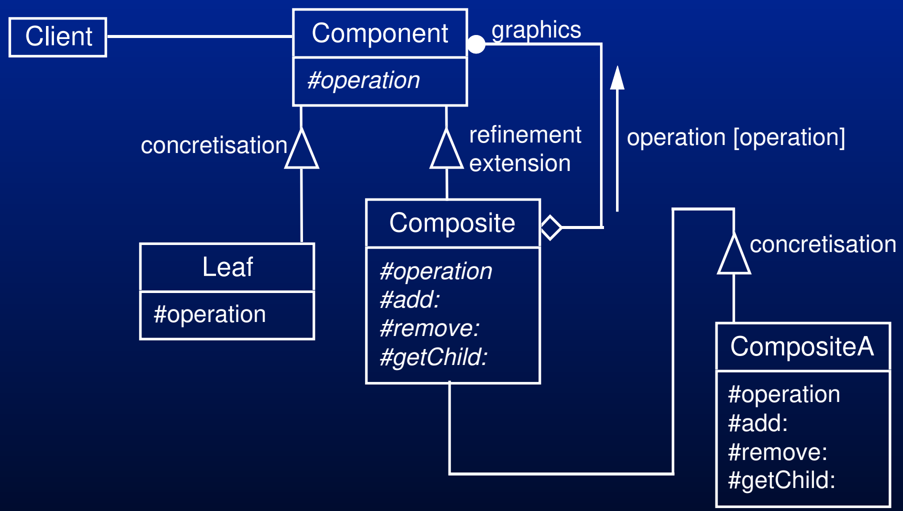
Overview

- Problems with reuse
- Problems with evolution
- What are reuse contracts?
- Reuse contracts at work
- Examining class hierarchies based on reuse contracts
- **Reuse contract research**
- Exercises: introduction to the browser

Reuse Contract Research

- Reuse contracts have been applied to
  - classes (inheritance)
  - sets of interacting classes/components
  - state diagrams
- Under investigation:
  - can reuse contracts describe design patterns?
  - generic reuse contracts
  - extraction of multi-class reuse contracts
  - software architectures and componentware
  - reuse contracts in a development environment
  - more documentation than interfaces and invocations

Design Pattern Example



Summary: Theory

- Reuse contracts <u>formally</u> document what a reuser can assume about a “reusable component”
- Reuse operators <u>formally</u> document how a reusable component is actually reused
- <u>Formal</u> rules for change propagation enable automatic detection of evolution conflicts

Summary: Practice

- Reuse contracts for inheritance can be extracted
  - examination of existing source code
  - understanding the design
  - human input is needed to filter out implementation details
  - bad coding style may give rise to extraction problems

Overview

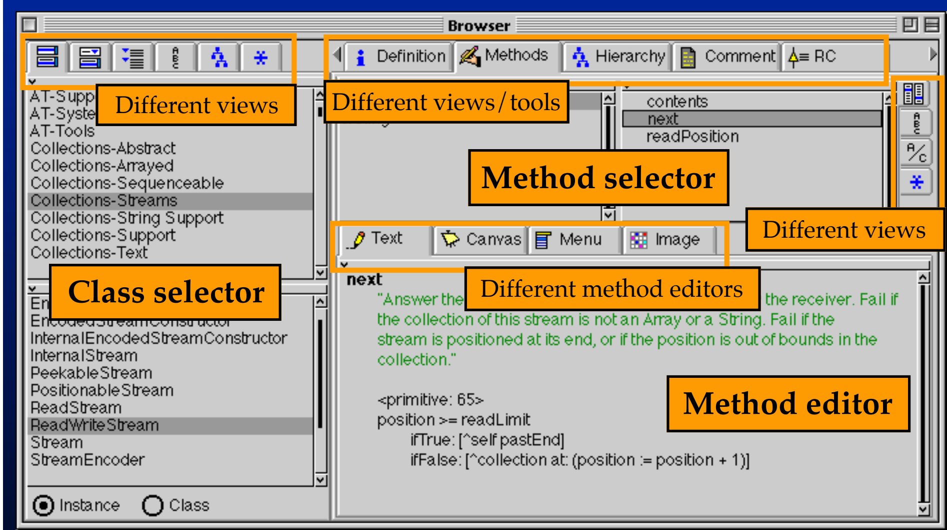
- Problems with reuse
- Problems with evolution
- What are reuse contracts?
- Reuse contracts at work
- Examining class hierarchies based on reuse contracts
- Reuse contract research
- **Exercises: introduction to the browser**

The Browser for the Exercises

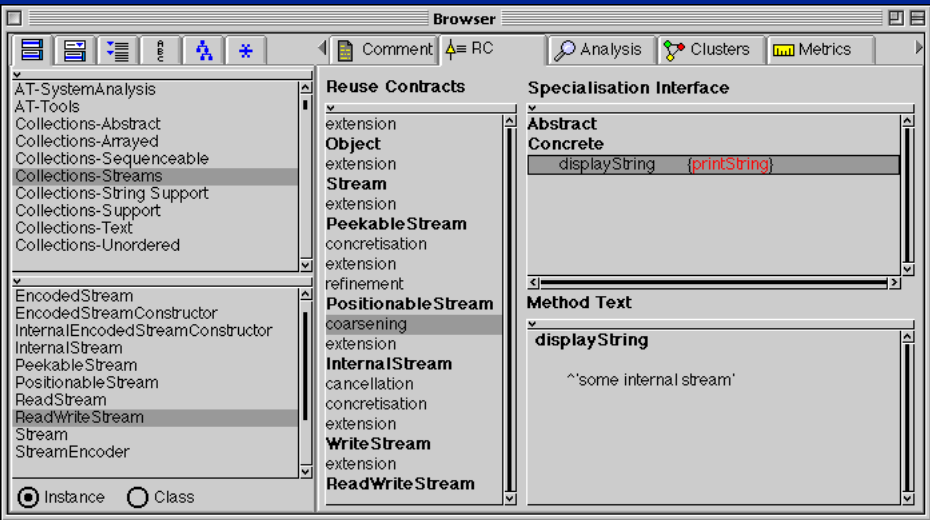
- Home-made fully-functional browser
- Composed of reusable “browser part components” built with ApplFLab
- Can easily be extended with other “class view/editor components”

See ESUG’96 Summer School “ApplFLab: Custom-made user interface components in VisualWorks”

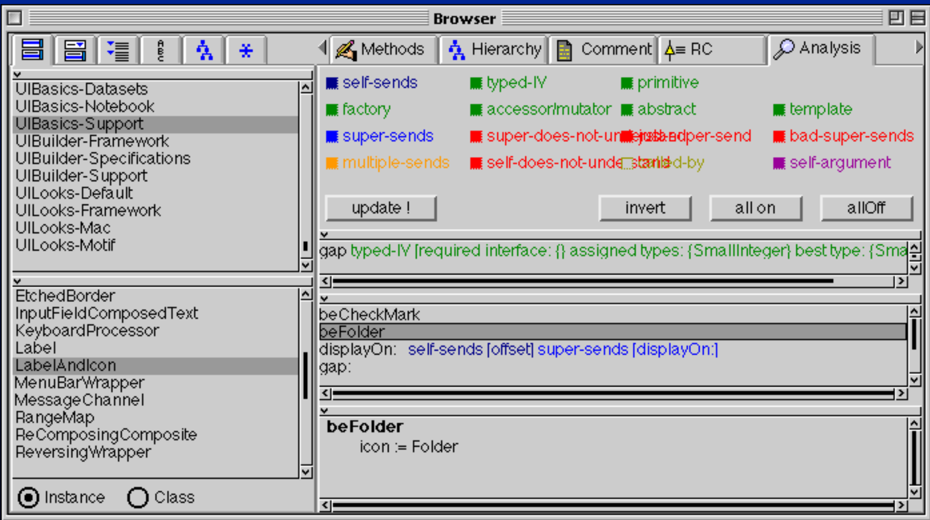
Enhanced Browser — General



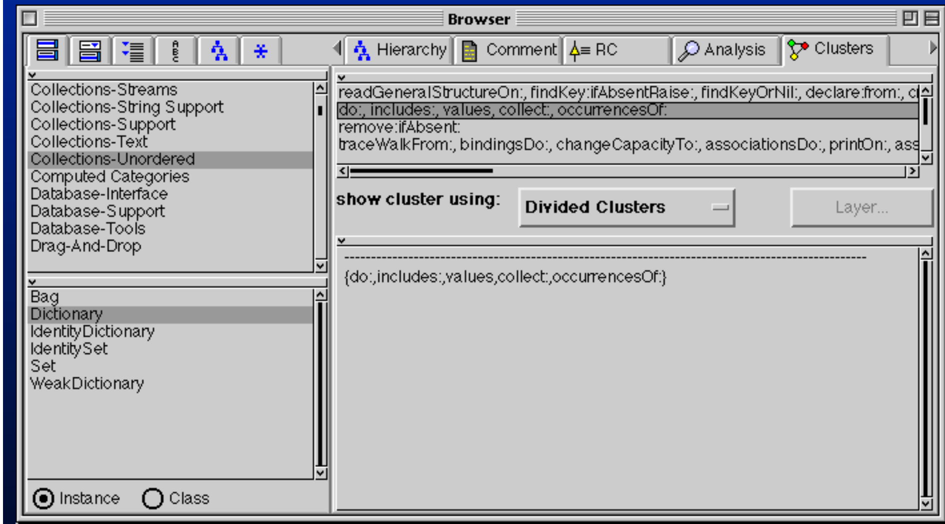
Browser — Reuse Contracts



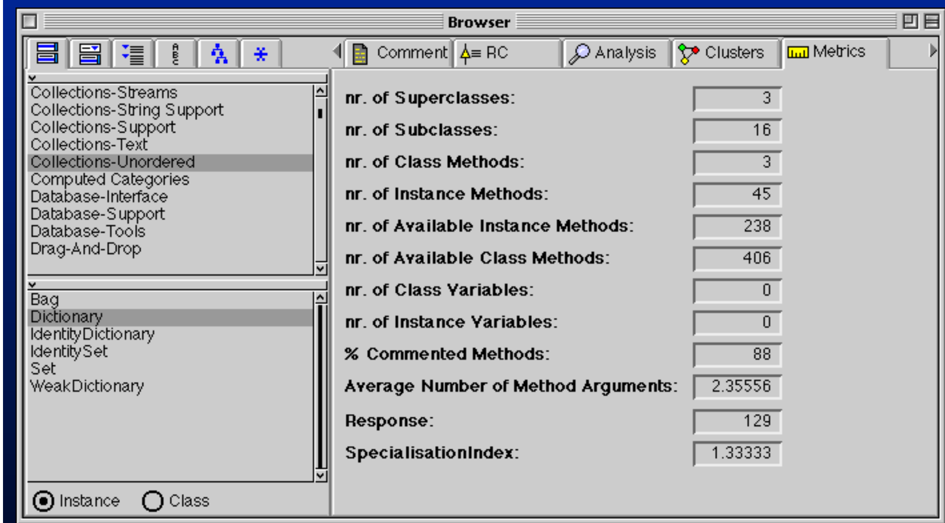
Browser — Code Analysis



Browser — Clusters



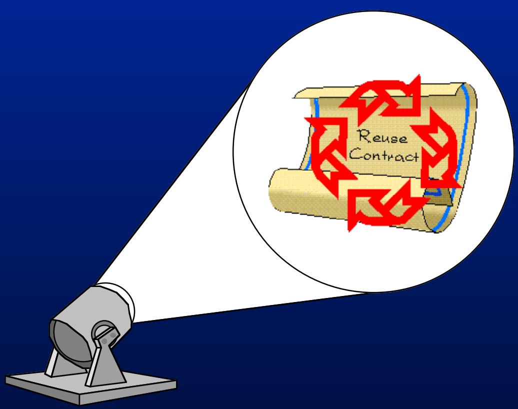
Browser — Metrics



Exercises

- Use the enhanced browser to investigate Smalltalk code
  - Examine class hierarchies based on extracted reuse contracts
  - Analyse the code to find methods that hinder reuse
  - Explore the different tools
- File in your own Smalltalk classes/ frameworks

Up-to-date Information



http://progwww.vub.ac.be/prog/pools/rcs/