GLASS, a Framework for Refactorings using FCA

Luca Scistri, Imen Benzarti, Petko Valtchev,
Yann-Gaël Guéhéneuc, Ghizlane El Boussaidi, Hafedh Mili

Version 1.0
25/04/14

Université Concordia University
GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science & Software Engineering

Outline

- Introduction
  - Context
  - Main Claim
  - Research Questions
- Background
  - FCA
  - Refactoring
- Feature Discovery
- Feature Refactoring
- Conclusion

Disclaimer

- Two parts
  - Descriptive
  - Prospective

Disclaimer

- Two parts
  - Descriptive
    - Hafedh Mili, Imen Benzarti, Amel Elkharraz, Ghizlane ElBoussaidi, Yann-Gaël Guéhéneuc, and Petko Valtchev ; Discovering Reusable Functional Features in Legacy Object-oriented Systems ; Transactions on Software Engineering, vol. 49, no. 7, pp. 3827–3856, IEEE CS Press, 2023
  - Prospective
    - Luca Scistri's on-going research work for his master thesis on refactorings class hierarchies using FCA and features

INTRODUCTION

Context

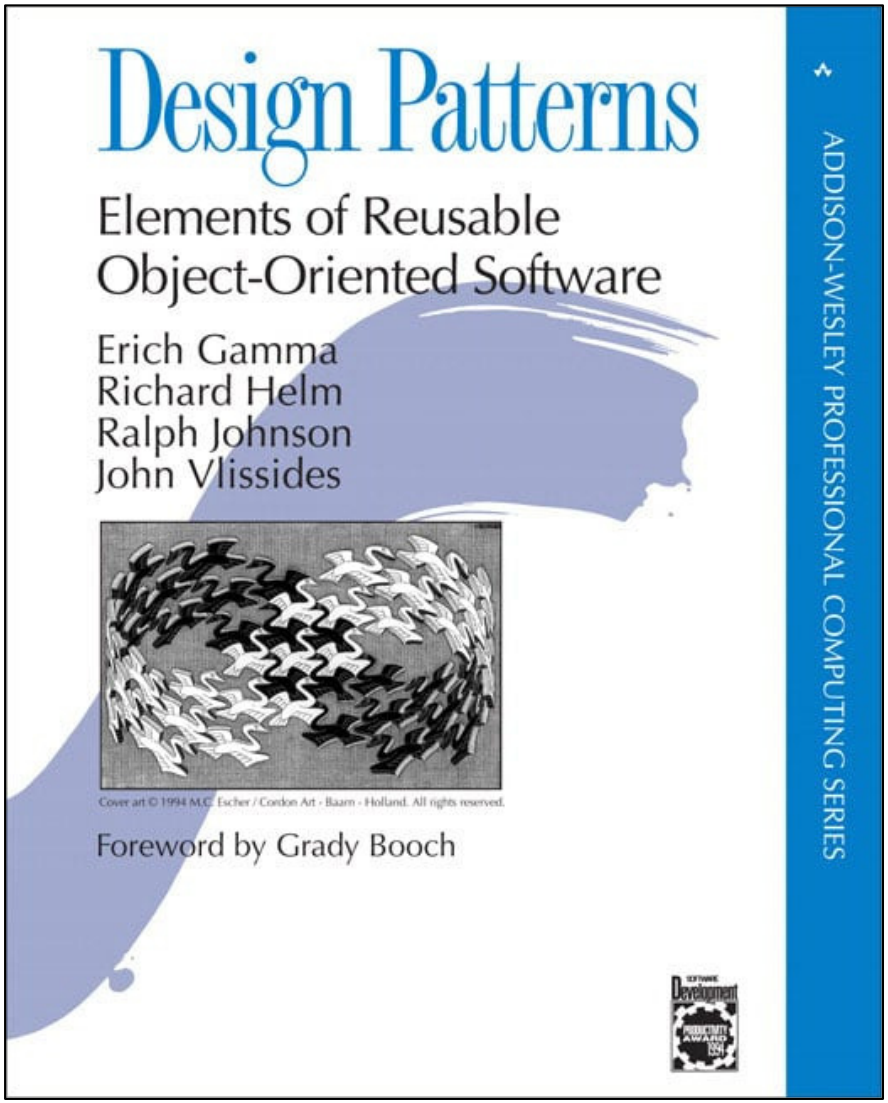
- OOP
  - Typically, Java
- Single inheritance of classes
- Multiple inheritance of interfaces

Context

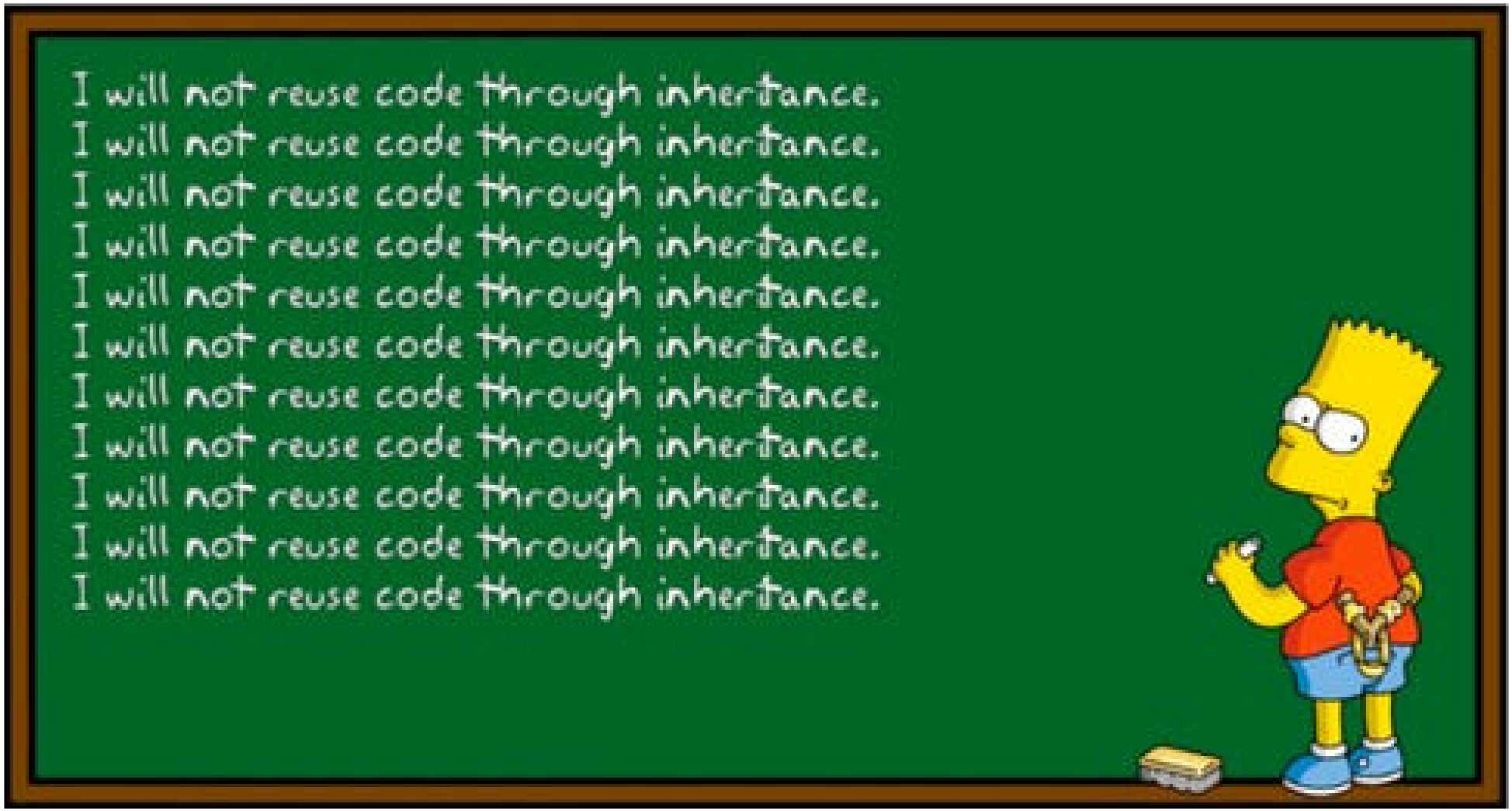
- Problem with Java (and others)
  - Inheritance plays two different roles
    - Typing
    - Reuse

Context

- Favour composition over inheritance
  - Well-known solution



Context

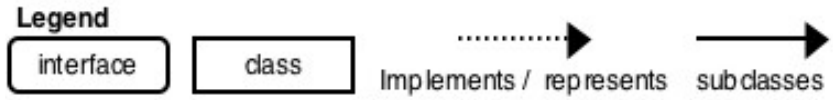

I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.
I will not reuse code through inheritance.

Main Claim

- We should distinguish completely inheritance and typing
- We should distinguish typing hierarchies from reuse hierarchies

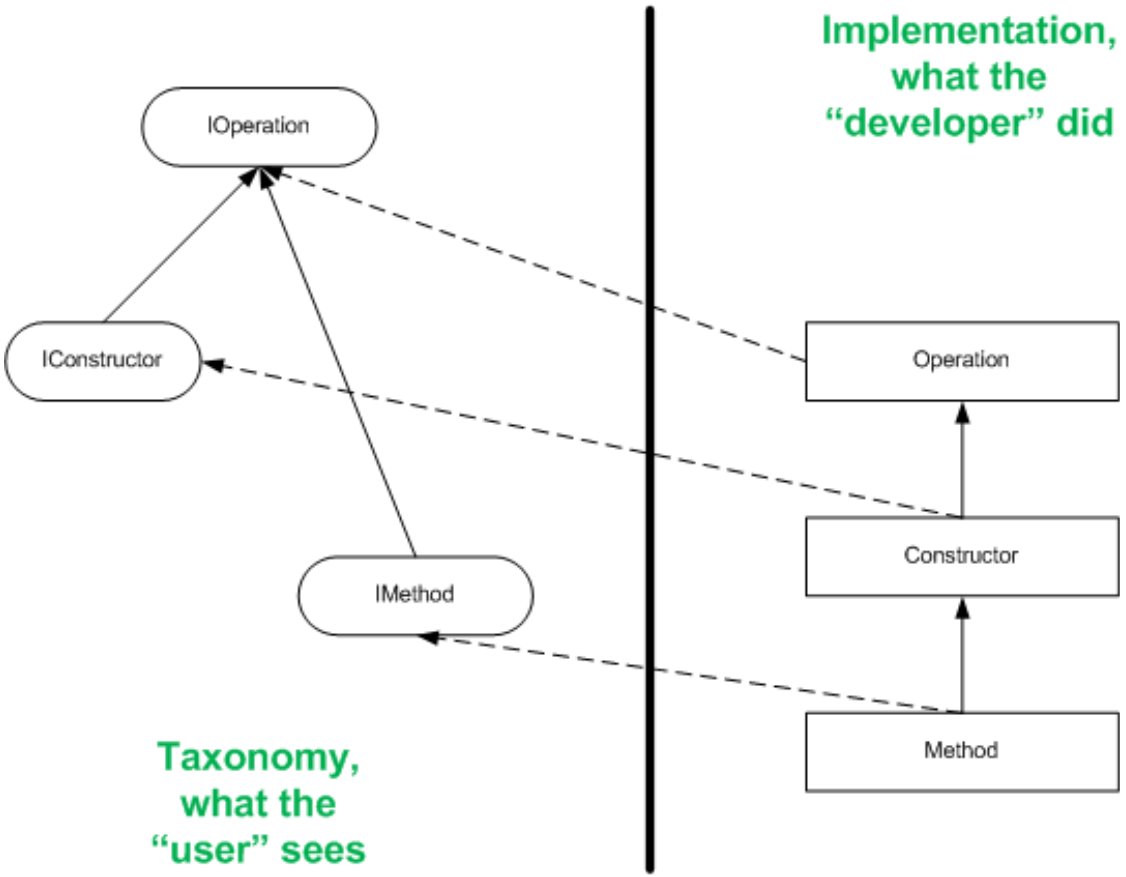
Main Claim

- We should distinguish **completely** inheritance and typing
- We should distinguish typing hierarchies from reuse hierarchies
  - **Fine-grained typing**
  - **Maximum reuse**



Main Claim

- Typing hierarchy
- Reuse hierarchy



Research Questions

- How to find typing/inheritance “misuses”?
- How to refactor these “misuses”?

BACKGROUND

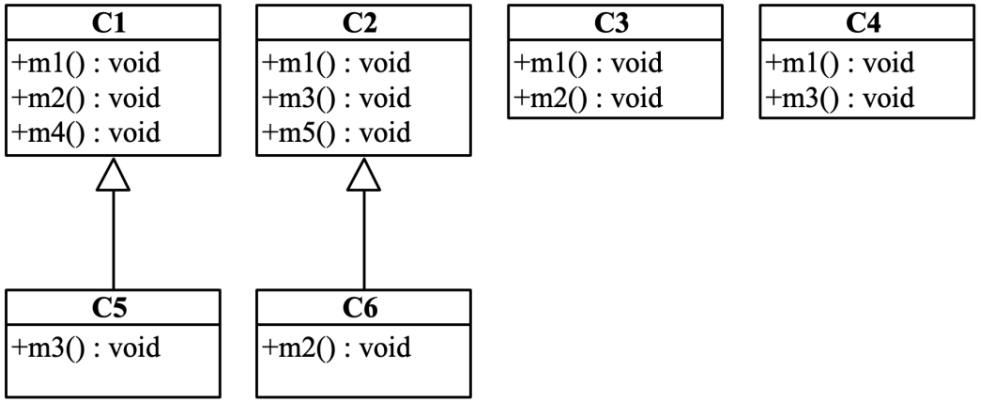
Background

- FCA
- Refactoring

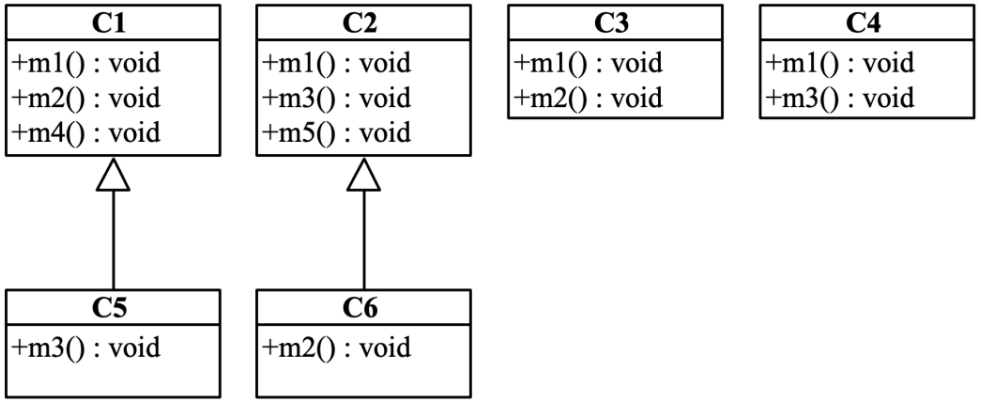
FCA

- Automatic classification technique
- Conceptual abstractions, or (formal) *concepts*, from individual elements
  - Based on their properties

FCA

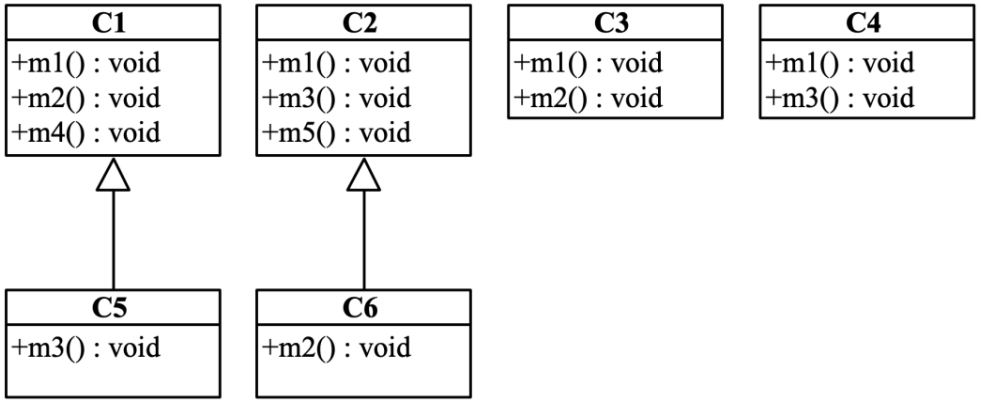

C1
+m1() : void
+m2() : void
+m4() : void
C2
+m1() : void
+m3() : void
+m5() : void
C3
+m1() : void
+m2() : void
C4
+m1() : void
+m3() : void
C5
+m3() : void
C6
+m2() : void

FCA

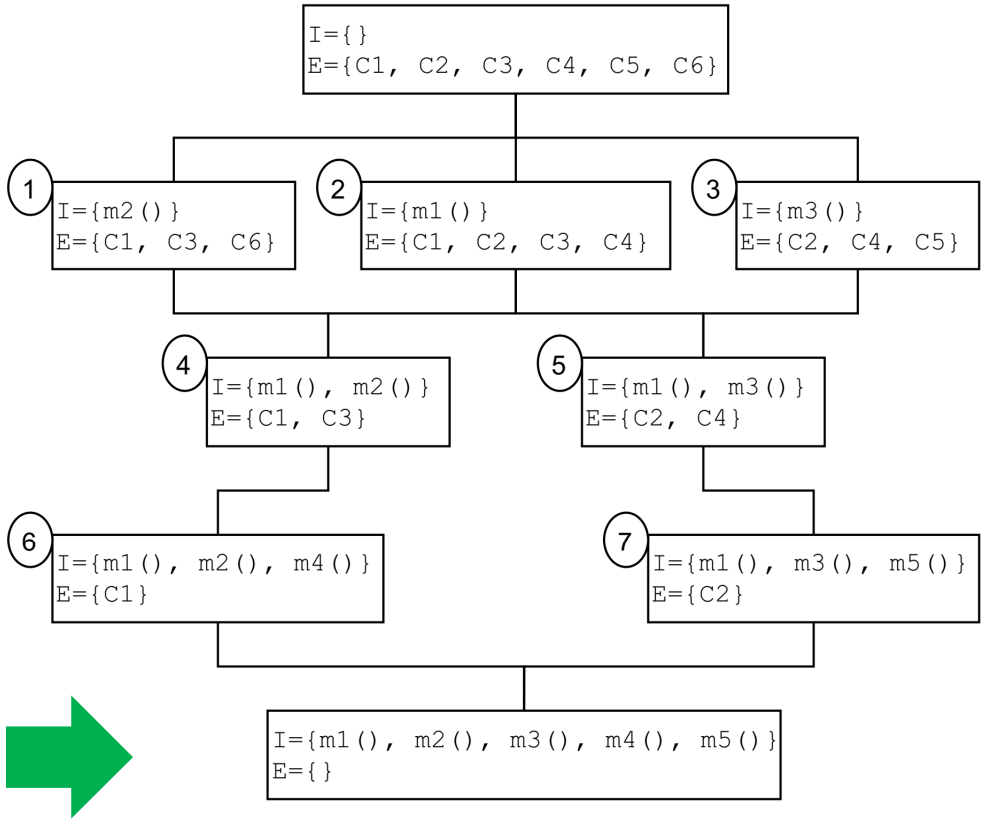


Classes \ Methods	m1 ()	m2 ()	m3 ()	m4 ()	m5 ()
C1	×	×		×	
C2	×		×		×
C3	×	×			
C4	×		×		
C5			×		
C6		×			

FCA



Classes \ Methods	m1 ()	m2 ()	m3 ()	m4 ()	m5 ()
C1	×	×		×	
C2	×		×		×
C3	×	×			
C4	×		×		
C5			×		
C6		×			



Refactoring

REFACTORING OBJECT-ORIENTED FRAMEWORKS

BY

WILLIAM F. OPDYKE

B.S., Drexel University, 1979
B.S., Drexel University, 1979
M.S., University of Wisconsin - Madison, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

- “[P]rogram restructuring operations”
- “[T]o be behavior preserving, provided that their preconditions are met”

Refactoring

- *Create classes*, *interfaces*
  - Including inserting them into existing hierarchies

- Extract class
- Pull Up Method/Field
- Push Down Method/Field

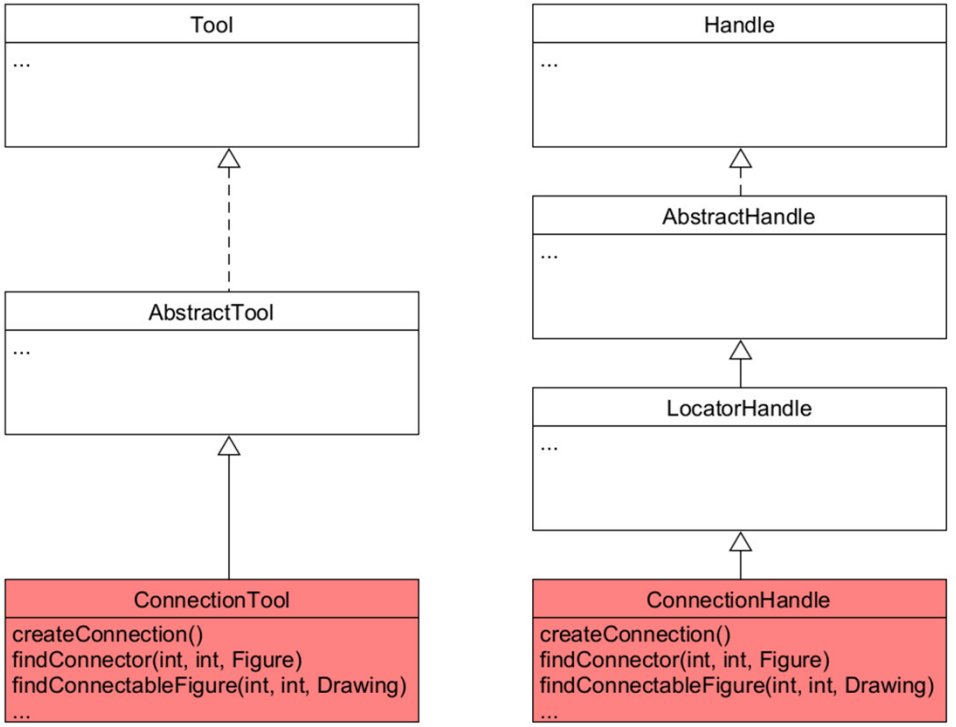
FEATURE DISCOVERY

Feature Discovery

- Examples
- Hypothesis
- Related Work
- Algorithm
- Evaluations
- Limitations

Examples

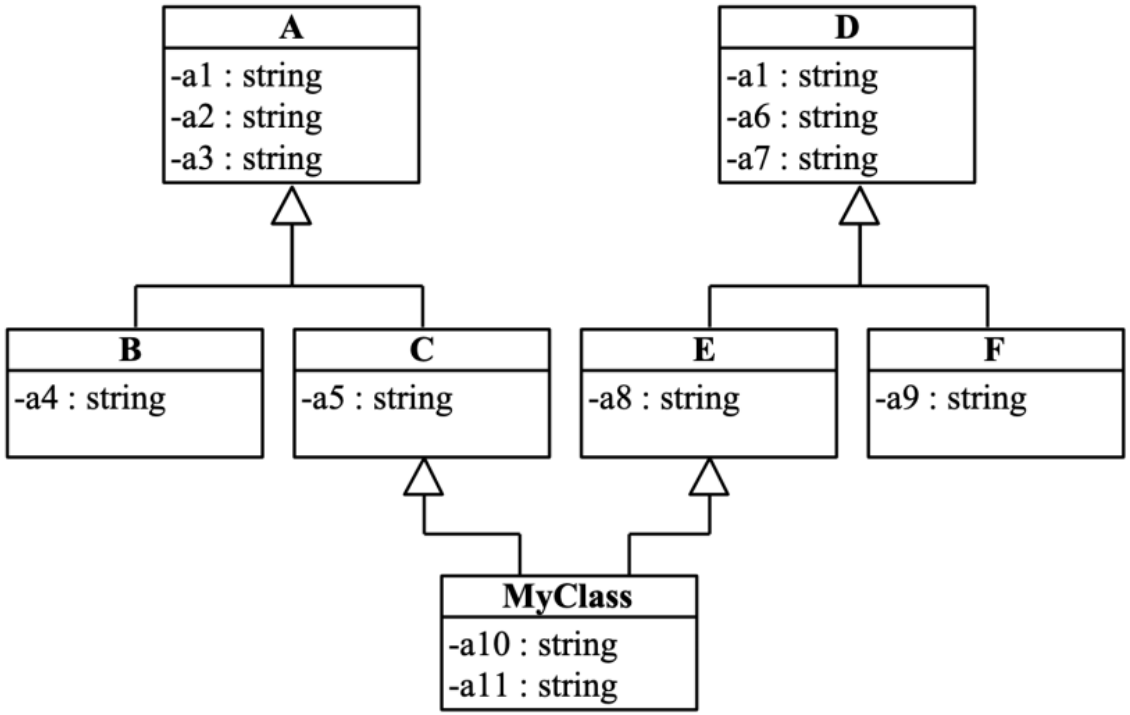
- Functional features
  - Functionally-cohesive and (relatively) self-contained domain functionalities



Examples

Multiple Inheritance

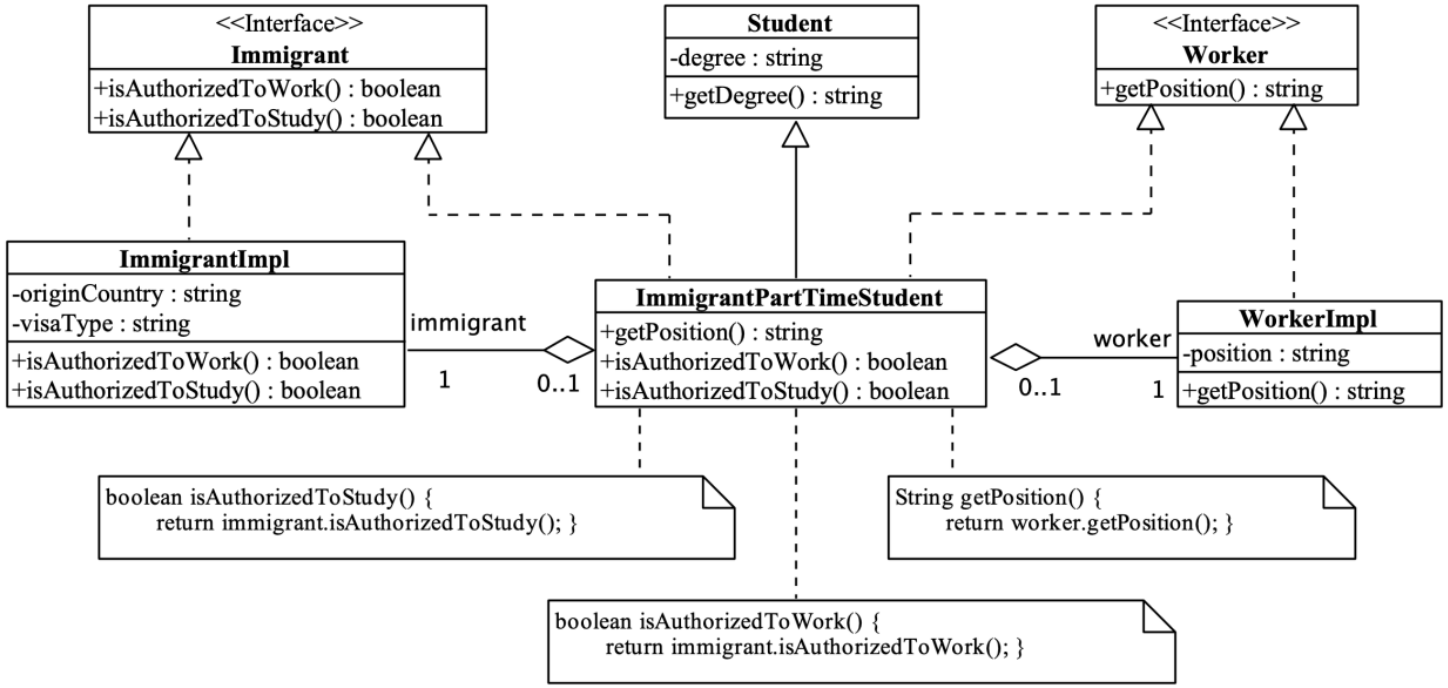
- Each functional feature is represented by its own class hierarchy
- A class combining several features inherits from these class hierarchies



Examples

Delegation

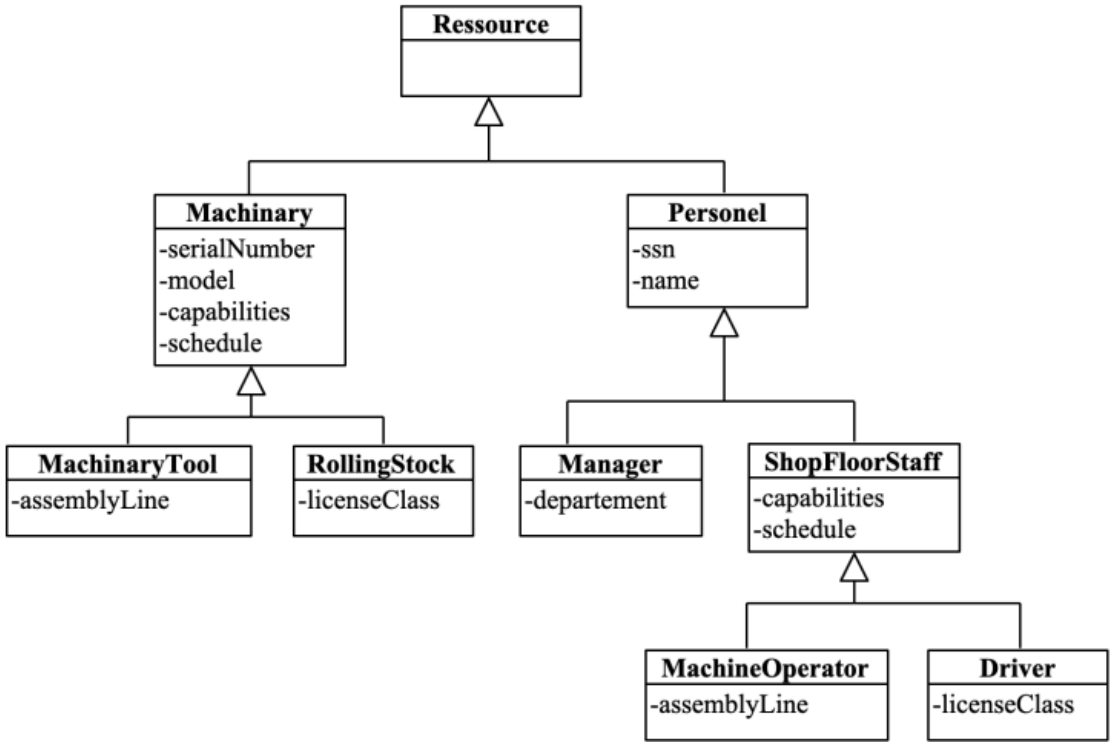
- Each functional feature is represented by its own class hierarchy
- A class combining several features aggregate their classes and delegate to their methods



Examples

Ad Hoc

- When developers missed useful/reusable features
- When the same feature is, inadvertently, duplicated several times



Hypothesis

- We can define and discover (ad hoc) functional features using FCA

Related Work

- Among others
  - Feature location
    - Thomas Eisenbarth, Rainer Koschke, and Daniel Simon ; Locating Features in Source Code ; Transactions on Software Engineering, vol. 29, no. 3, IEEE CS Press, 2003
  - Feature discovery
    - Paul W. McBurney, Cheng Liu, and Collin McMillan ; Automated Feature Discovery via Sentence Selection and Source Code Summarization ; Software Evolution and Process, vol. 28, no. 2, Wiley, 2016

Algorithm

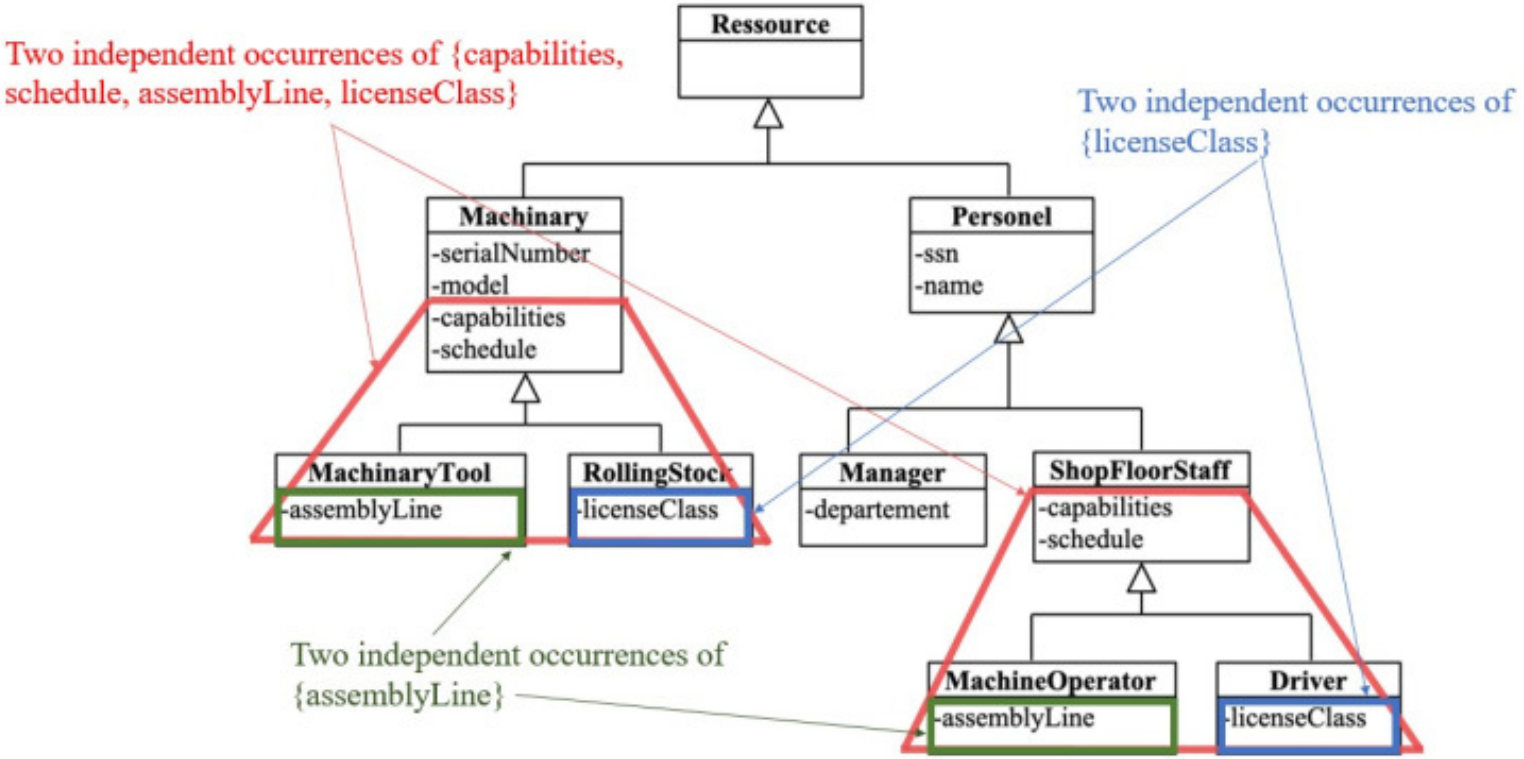
- Relatively straightforward
  - Multiple inheritance
  - Delegation

Algorithm

- Reverse inheritance
  - Incidence relationship associating a `class` with the union of the elements of its sub-classes

Algorithm

- Reverse inheritance
  - Incidence relationship associating a `class` with the union of the elements of its sub-classes



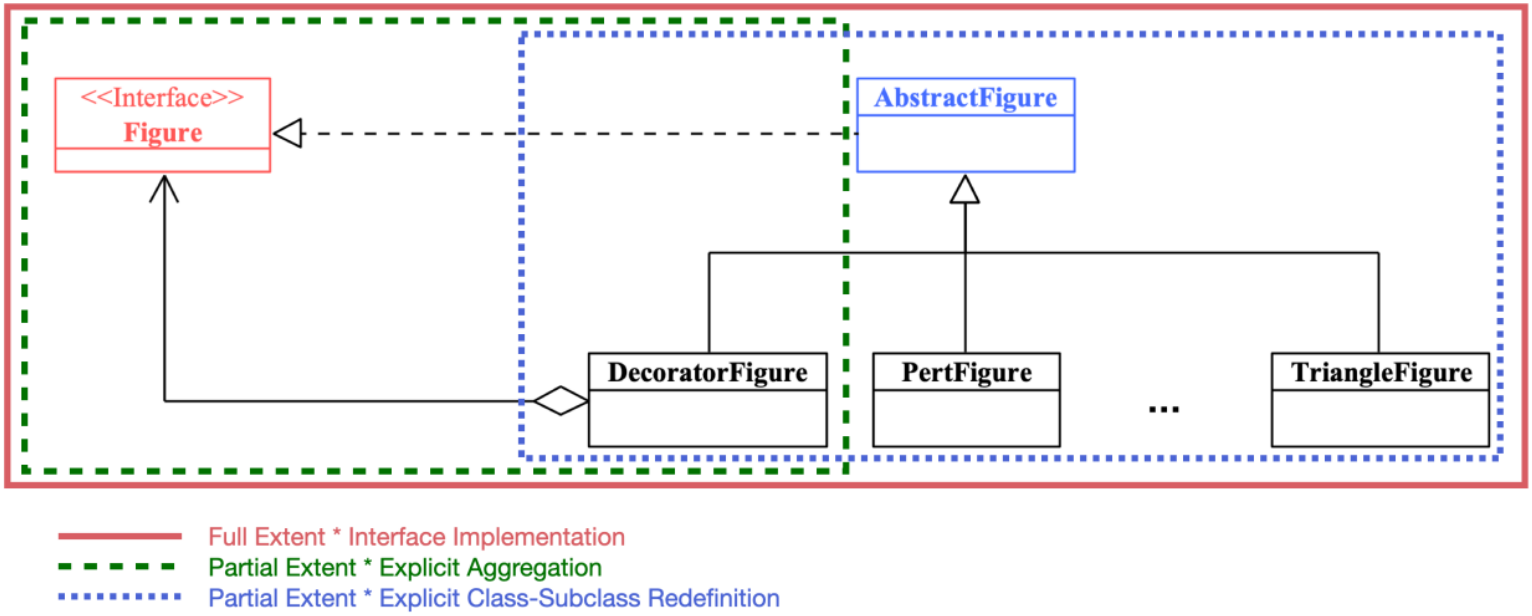
Algorithm

- Reverse inheritance
  - Incidence relationship associating a `class` with the union of the elements of its sub-classes

Input: concept lattice $\mathcal{L}$
Output: feature candidates *FeatureList*
ListConcept $\leftarrow$ children($\top_{\mathcal{L}}$)
FeatureList $\leftarrow \emptyset$
while *ListConcept* $\neq \emptyset$ **do**
  $(X, Y) \leftarrow$ extract($ListConcept$)
  if $|\mathbf{min}(X)| > 1$ **then**
    add $((X, Y)$, *FeatureList*)
    foreach $(X', Y') \in children((X, Y))$ **do**
      add $((X', Y')$, *ListConcept*)
      if $(|\mathbf{min}(X')| = |\mathbf{min}(X)|)$ **then**
        remove $((X, Y)$, *FeatureList*)
    end
  end
end

Algorithm

- Reverse inheritance
  - Incidence relationship associating a `class` with the union of the elements of its sub-classes



Evaluations

- Quantitative
  - Precision
  - No recall
- Qualitative
  - Manual
- Comparison

Evaluations

Quantitative

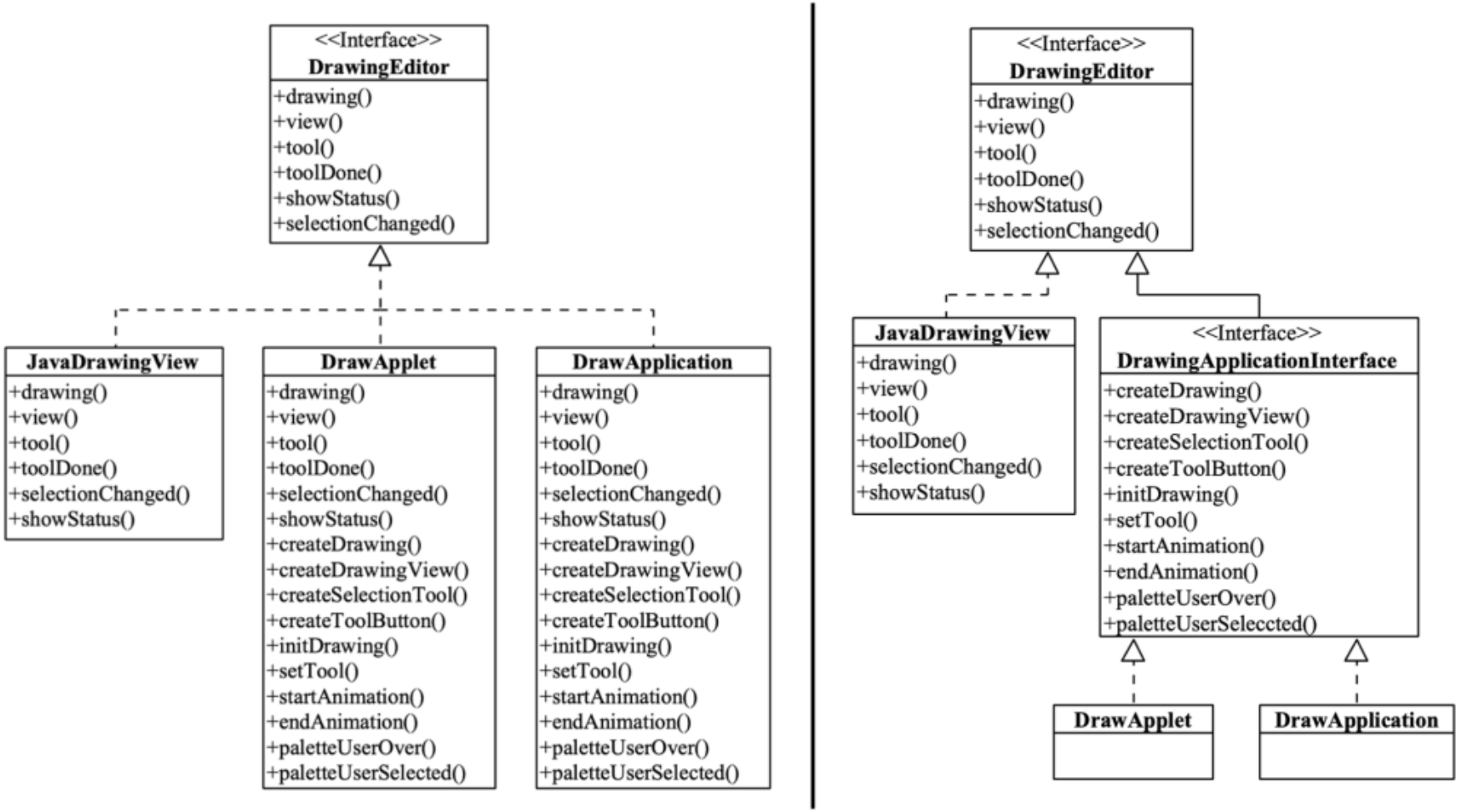
Systems	#CF	#AD	#INT	#SUB	#AGR	#PART
FreeMind	69	32	1	6	6	24
JavaWebMail	50	23	0	5	7	15
JHotDraw	154	54	41	18	0	41
JReversePro	26	24	1	1	0	0
Lucene	91	47	4	23	4	13

CF: candidate functional features
AD: *Ad Hoc* features
INT: *Full Behavior* of *Full Extent* of *Explicit Interface Implementation*
SUB: *Full Behavior* of *Full Extent* of *Explicit Class Subclass Redefinition*
AGR: *Full Behavior* of *Full Extent* of *Explicit Aggregation*
PART: *Partial Extent* not included in previous categories

Systems	%AD	%INT	%SUB	%AGR	%PART
FreeMind	46.38	1.45	8.7	8.7	34.78
JavaWebMail	46	0	10	14	30
JHotDraw	35.06	26.62	11.69	0	26.62
JReversePro	92.3	3.8	3.8	0	0
Lucene	51.65	4.4	25.27	4.4	14.3

Evaluations

Qualitative



Evaluations

Comparison

- Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng ; Service Candidate Identification from Monolithic Systems based on Execution Traces ; Transactions on Software Engineering, vol. 47, no. 5, pp. 987–1007, IEEE CS Press, 2021

#	Type	Extent and Intent	#	Entities
16	*Partial Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.CategoryMapper **....service.CatalogService** **....web.actions.CatalogActionBean** getCategoryList()	SC0	....domain.Category **....service.CatalogService** **....web.actions.CatalogActionBean** ....domain.Product ....domain.Item ....domain.Sequence
25	*Full Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.OrderMapper **....service.OrderService** getOrdersByUsername(String) getOrder(int) insertOrder(Order)	SC1	....domain.LineItem ....web.actions.OrderActionBean **....service.OrderService** ....domain.Order
			SC2	....domain.Cart ....domain.CartItem ....web.actions.CartActionBean
31	*Full Extent* *Full Behaviour* *Explicit Aggregation*	....mapper.AccountMapper **....service.AccountService** insertAccount(Account) updateAccount(Account)	SC3	**....service.AccountService** ....web.actions.AccountActionBean ....domain.Account

Limitations

- Threats to validity
  - Construct: what constitutes an interesting or useful functional feature?
  - Internal: are we reliable judges of the usefulness of the found functional features?
  - External: can the results be generalised to other programs? Languages?
- No suggestions of refactorings

Limitations

- Threats to validity
  - Construct: what constitutes an interesting or useful functional feature?
  - Internal: are we reliable judges of the usefulness of the found functional features?
  - External: can the results be generalised to other programs? Languages?

- **No suggestions of refactorings**

FEATURE REFACTORING

Feature Refactoring

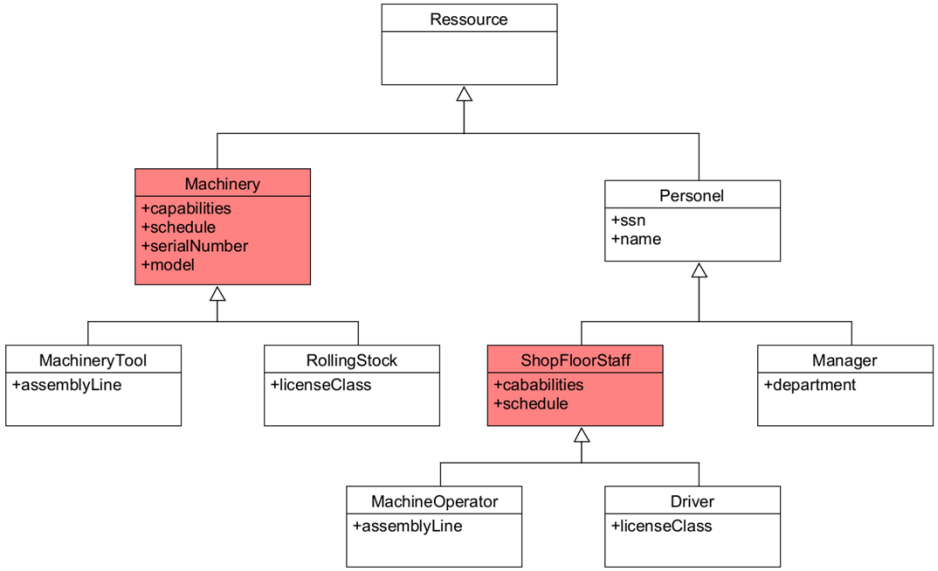
- Examples
- Hypothesis
- Related Work
- Algorithm
- Limitations
- Discussions

Examples

- Ad hoc feature

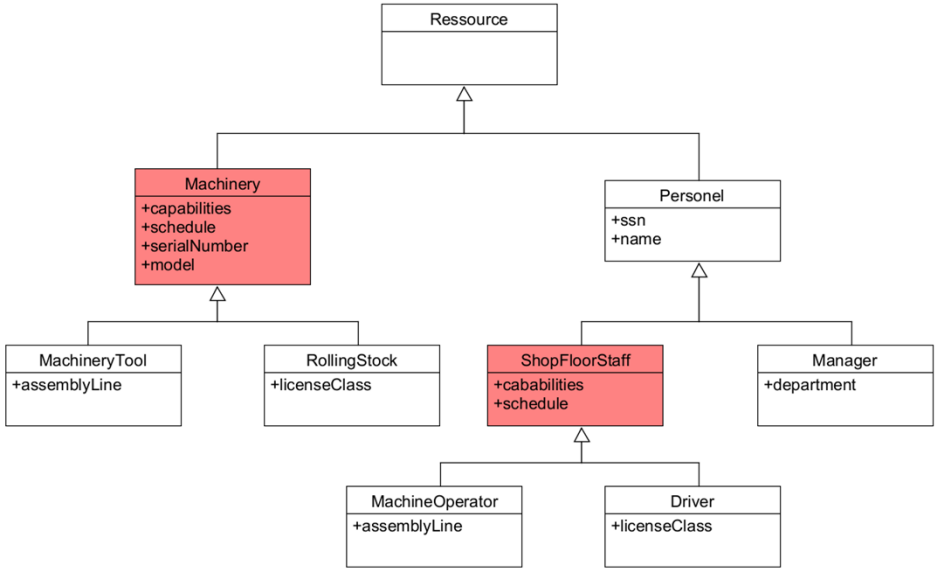
Examples

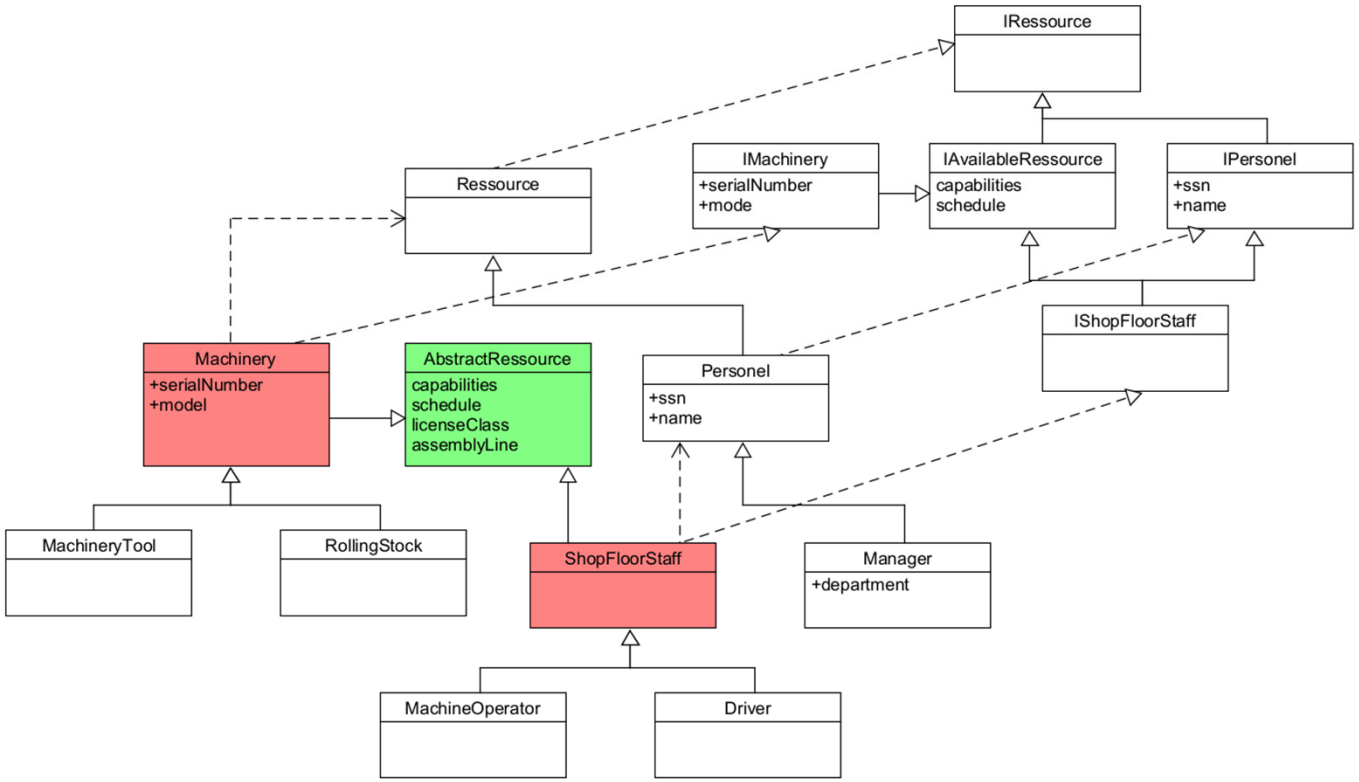
- Ad hoc feature



Examples

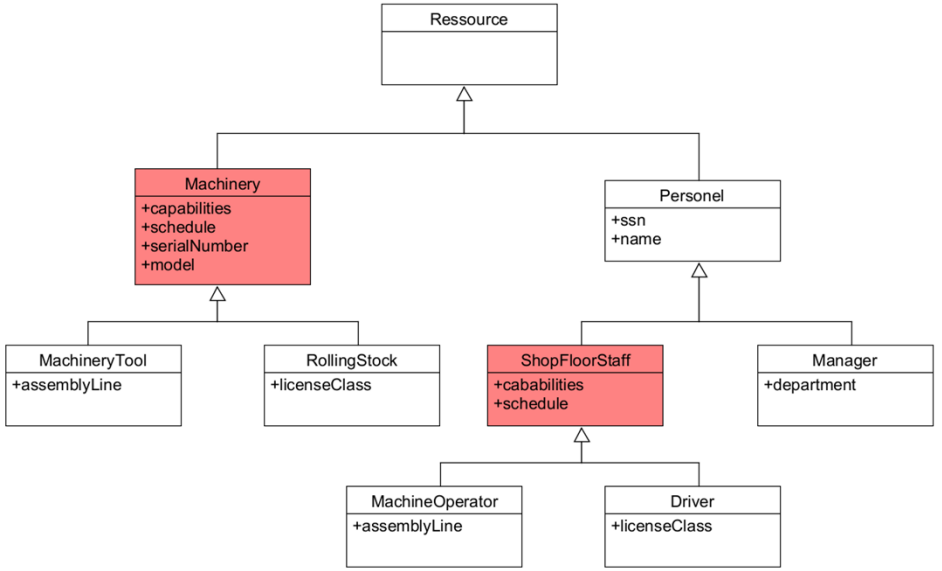
- Ad hoc feature

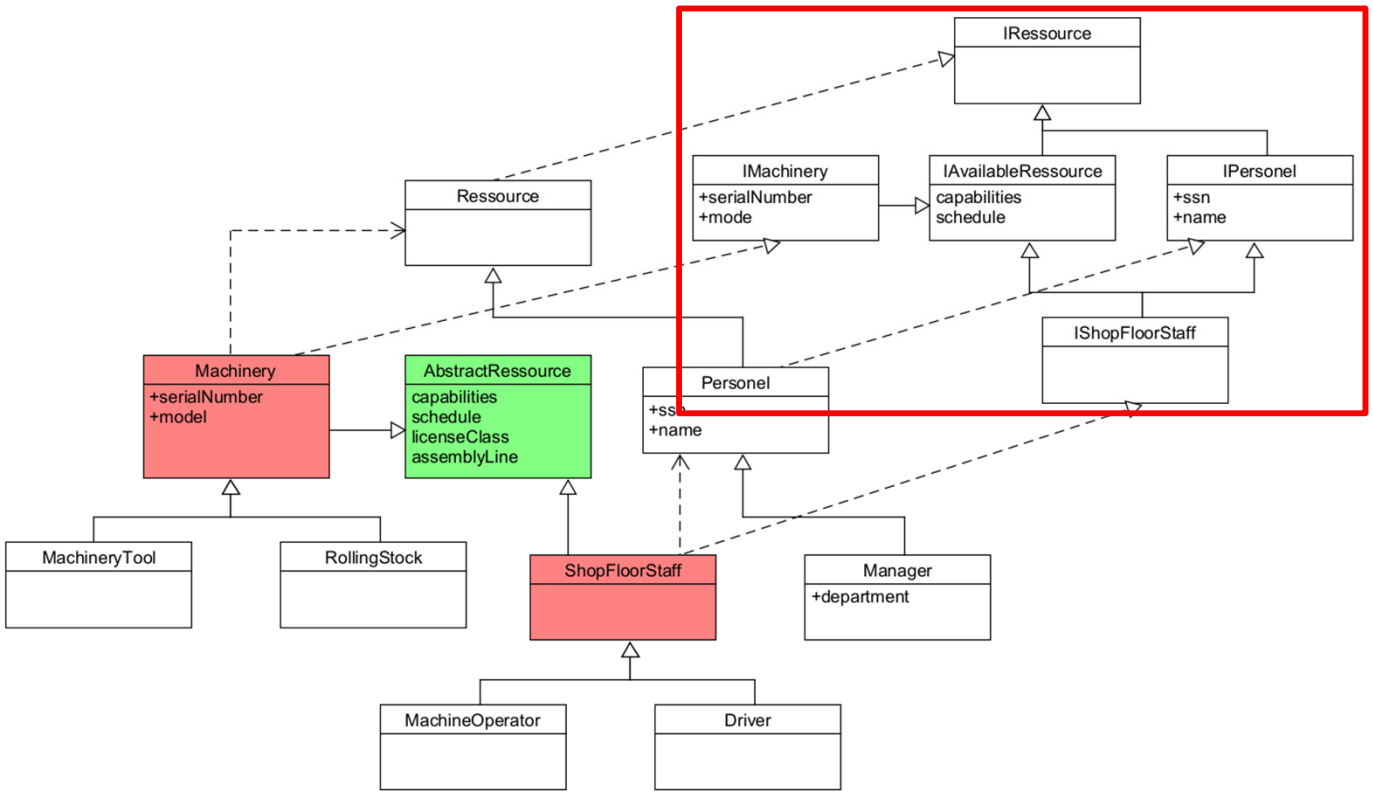




Examples

- Ad hoc feature



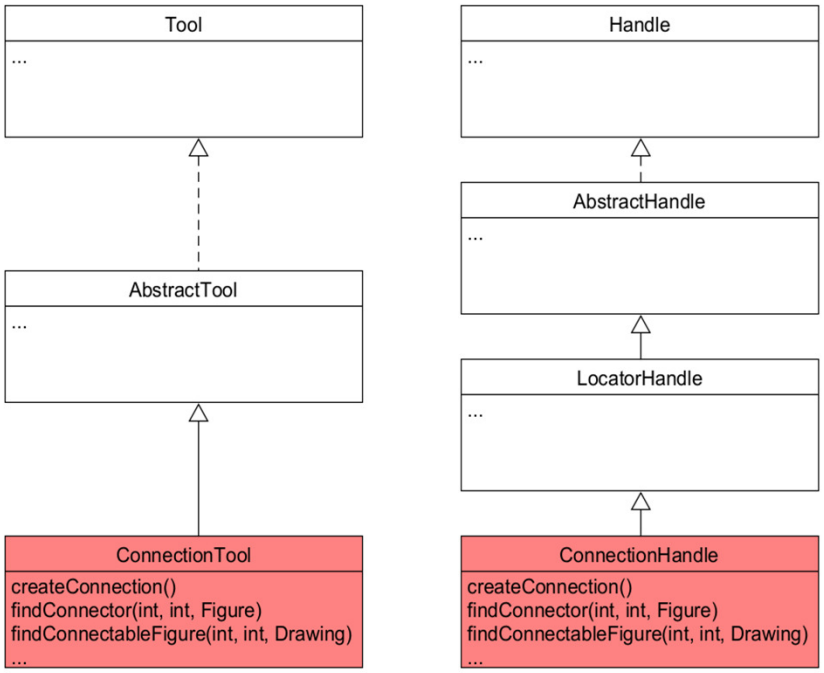


Examples

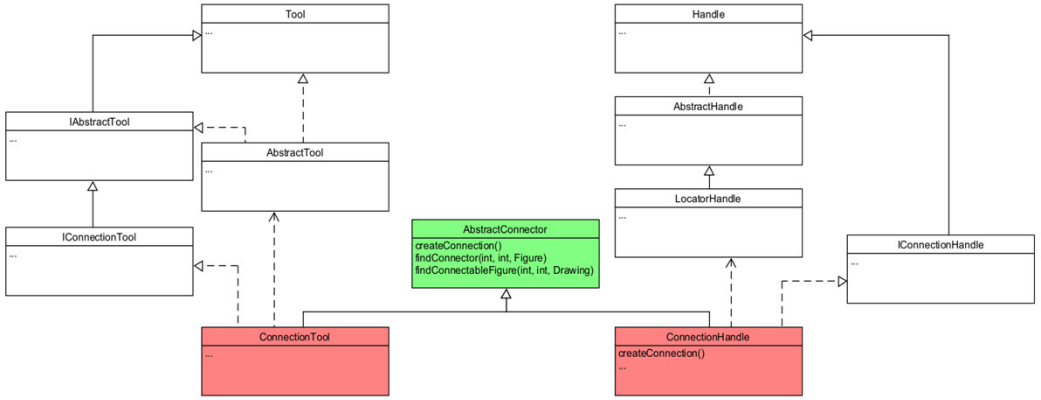
- JHotDraw v5.2
  - In `CH.ifa.draw.standard`
- JavaWebMail v0.7
  - In `net.wastl.webmail.xml`
- PADL Metamodel
  - In `padl.kernel` and `padl.kernel.impl`

Examples

JHotDraw



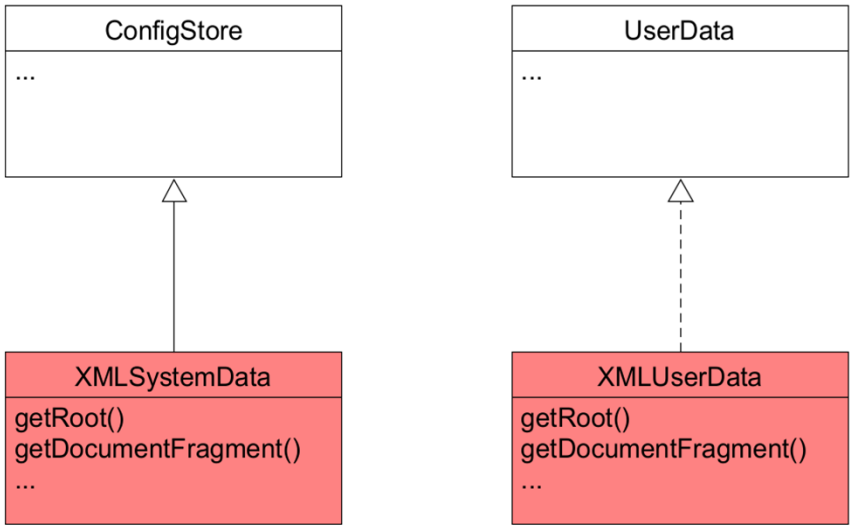
`CH.ifa.draw.standard`

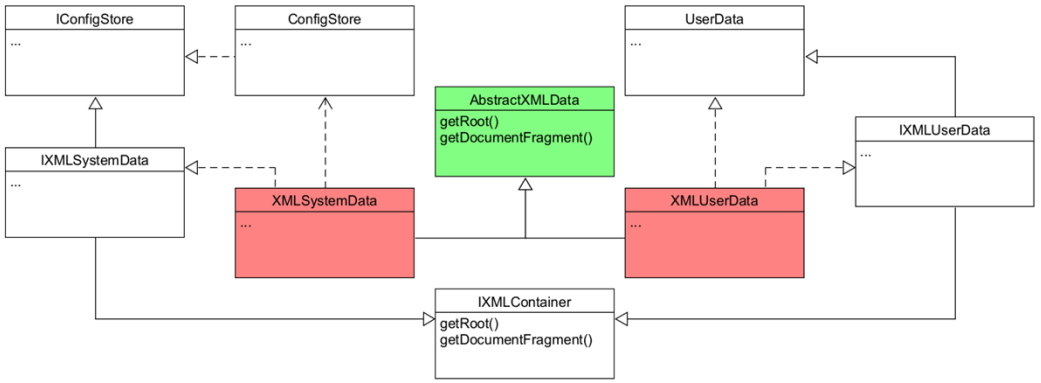


Examples

JavaWebMail

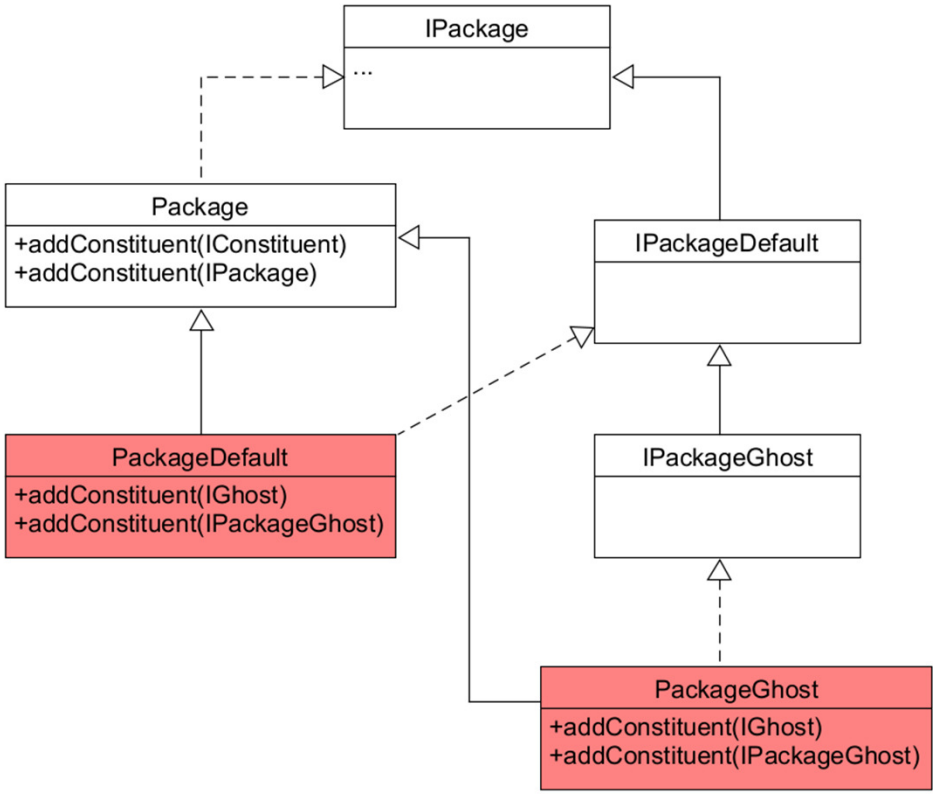
`net.wastl.webmail.xml`



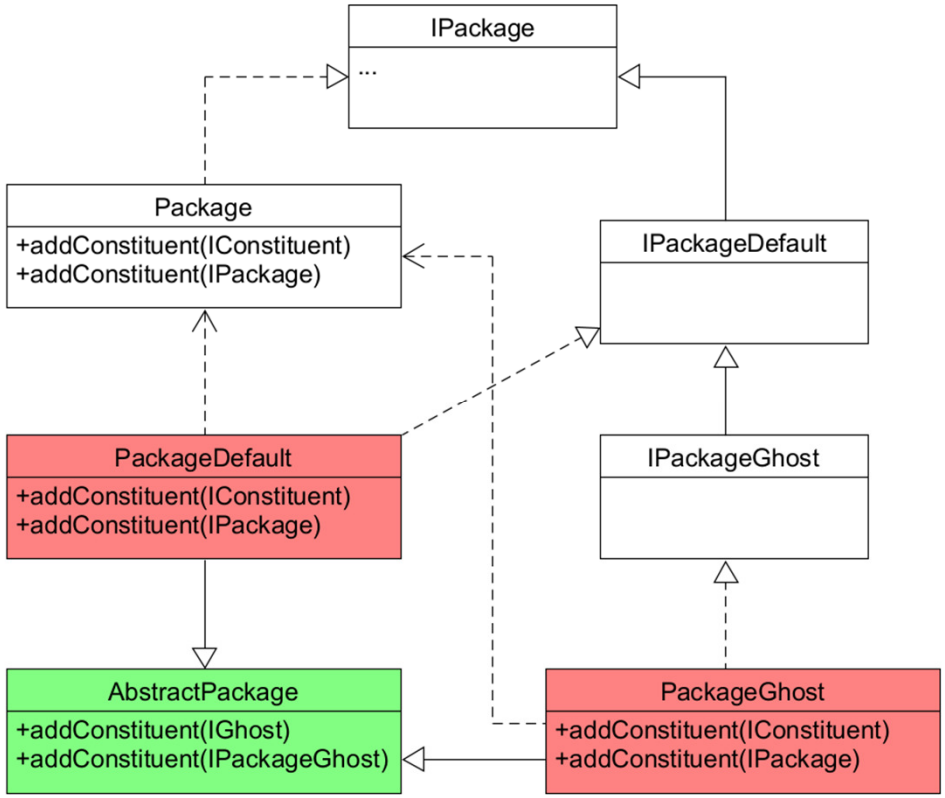


Examples

PADL



`padl.kernel` and `….impl`



Hypothesis

- We can use discovered feature and FCA (again) to refactor hierarchies

Related Work

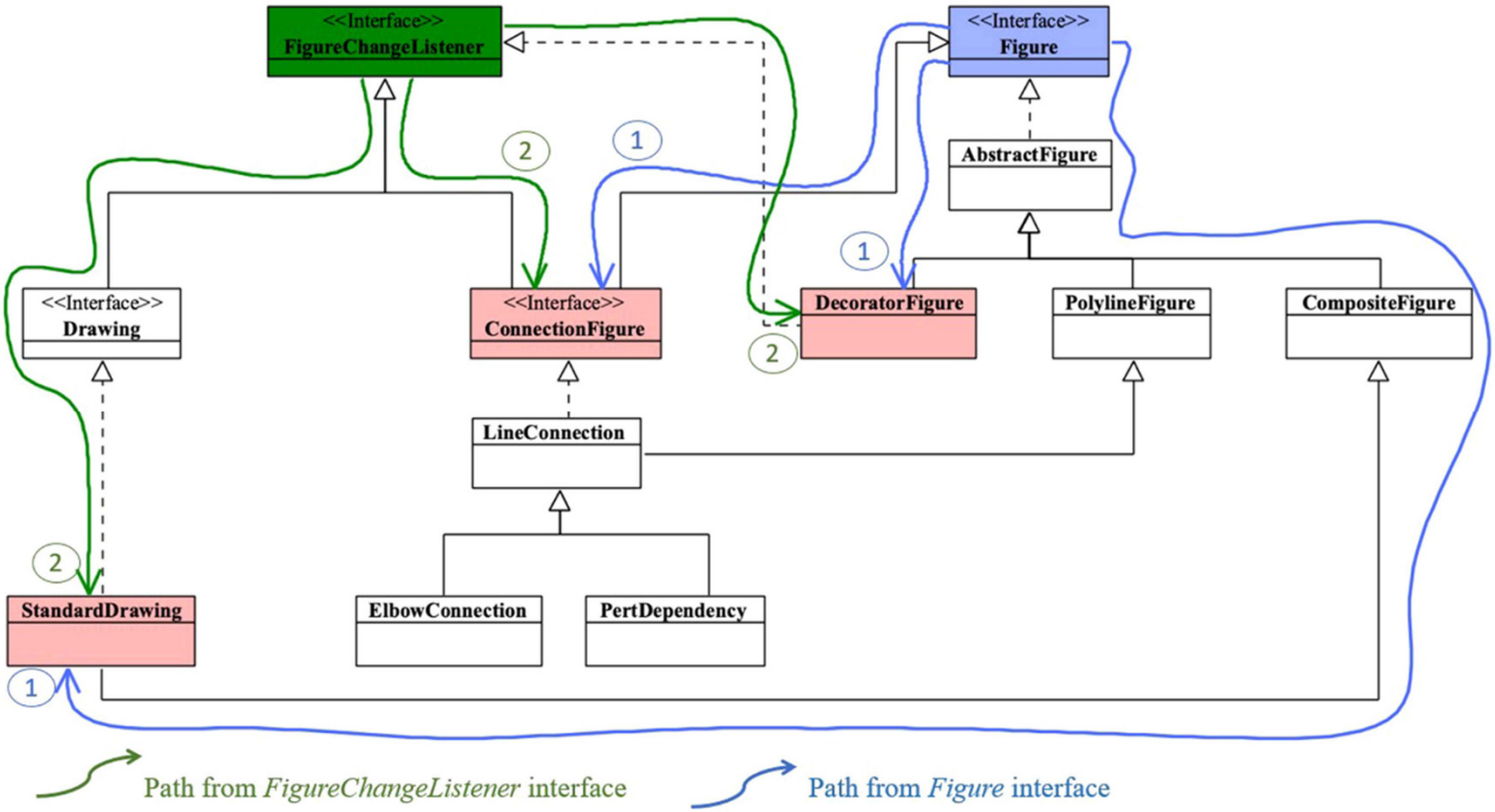
- Among others
  - Replacing inheritance with delegation
    - Hannes Kegel and Friedrich Steimann ; Systematically Refactoring Inheritance to Delegation in Java ; Proceedings of the 13th International Conference on Software Engineering, ACM Press, 2008
  - Using FCA to improve type hierarchies
    - Marianne Huchard and Hervé Leblanc ; Computing Interfaces in Java ; Proceedings of the 15th International Conference on Automated Software Engineering, IEEE CS Press, 2000
    - Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc ; Refactorings of Design Defects using Relational Concept Analysis ; Proceedings of the 6th International Conference on Formal Concept Analysis, Springer, 2008

Algorithm

- Seven steps
  1. Choosing where the refactoring takes place
  2. Extracting interfaces from the hierarchy
  3. Using FCA to create new type hierarchy
  4. Creating a new class
  5. Replacing inheritance with delegation (optional)
  6. Making the new class a superclass of the classes in the extent
  7. Pulling up methods in the intent to the class

Algorithm

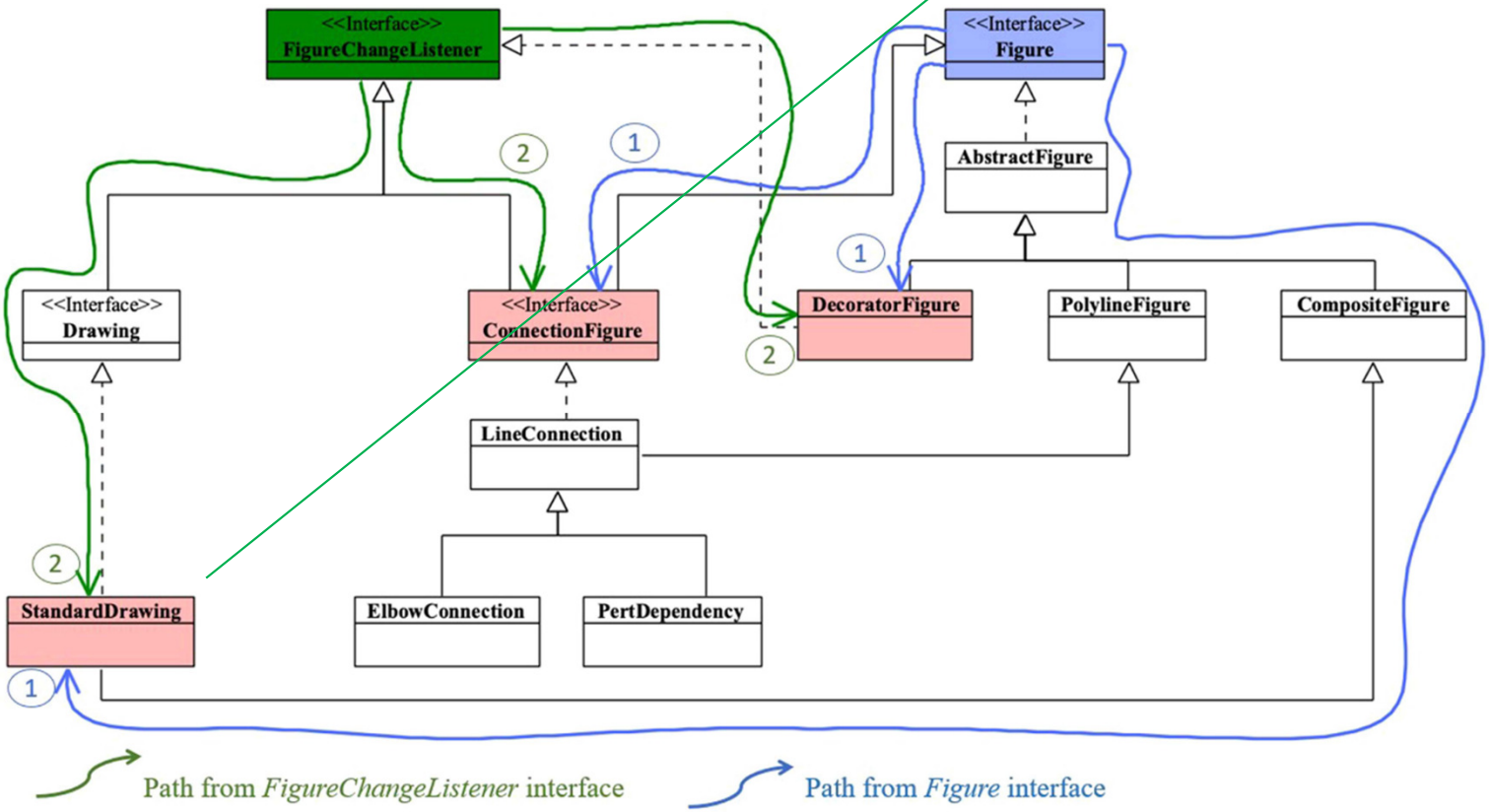
1. Choosing the feature to refactor is difficult
   - False positives
   - Ad hoc features *descending* from deliberate ones



Algorithm

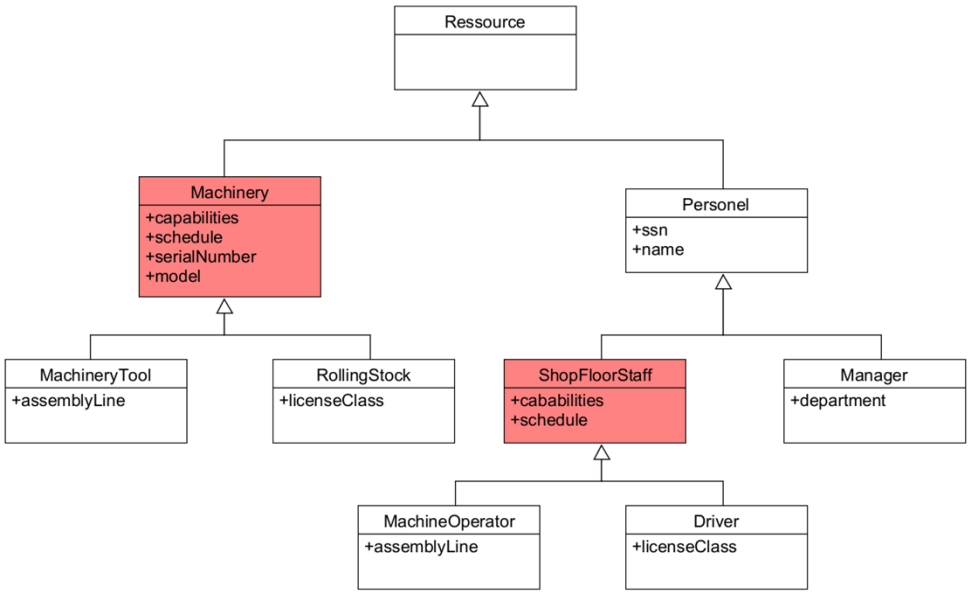
Through reverse inheritance, the methods from this class are recognized as two independent occurrences

1. Choosing the feature to refactor is difficult
   - False positives
   - Ad hoc features *descending* from deliberate ones



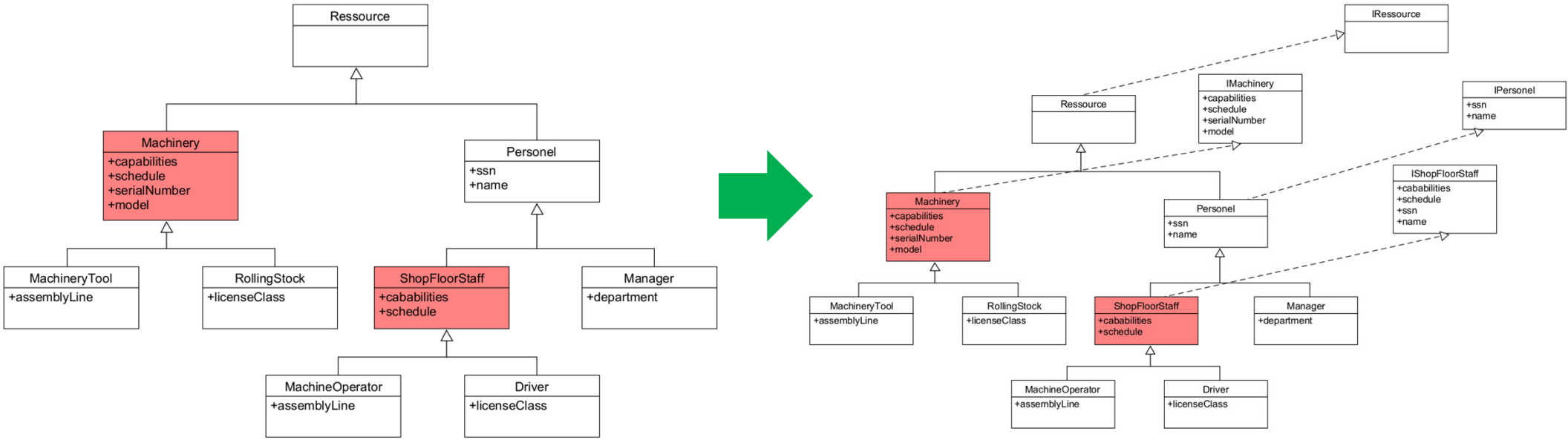
Algorithm

2. Once a feature is chosen, we extract an interface from each (sub)type of the extent



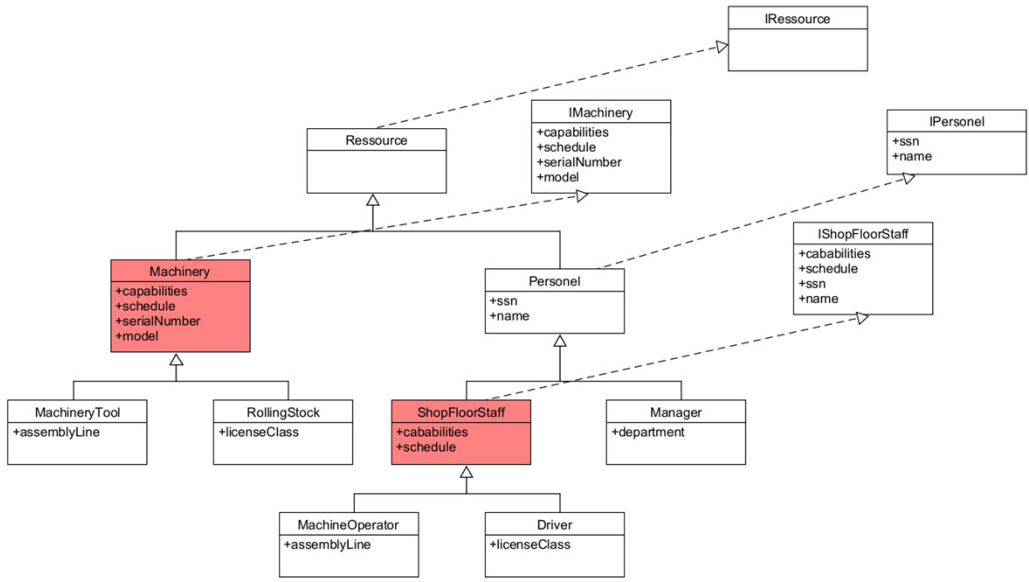
Algorithm

2. Once a feature is chosen, we extract an interface from each (sub)type of the extent



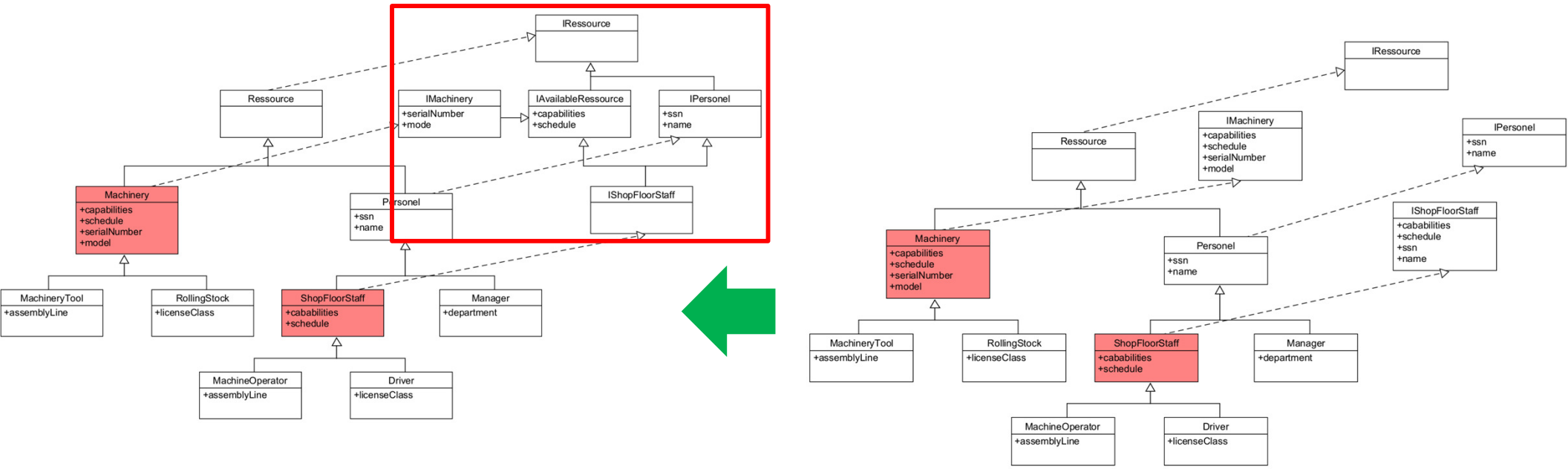
Algorithm

3. We use FCA to fix the hierarchy



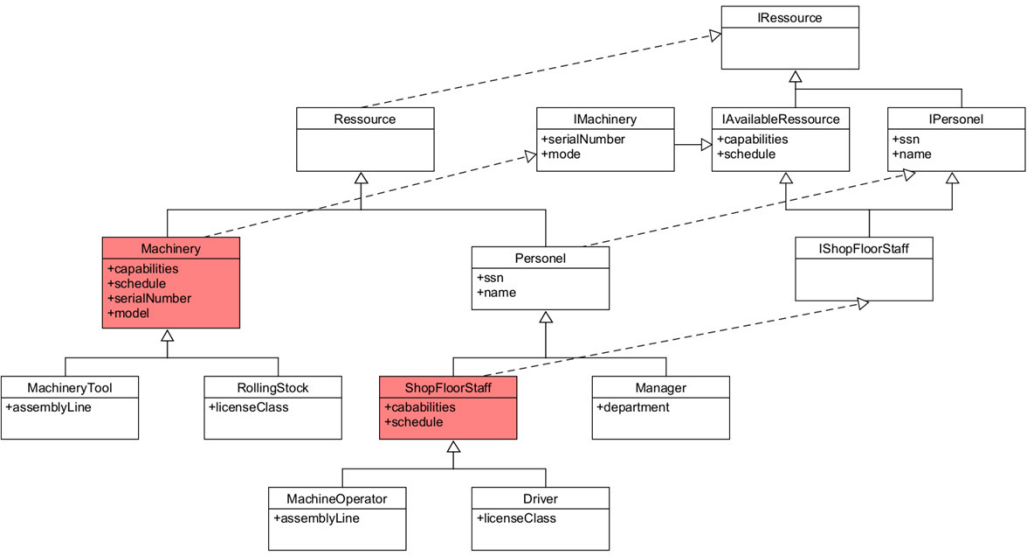
Algorithm

3. We use FCA to fix the hierarchy



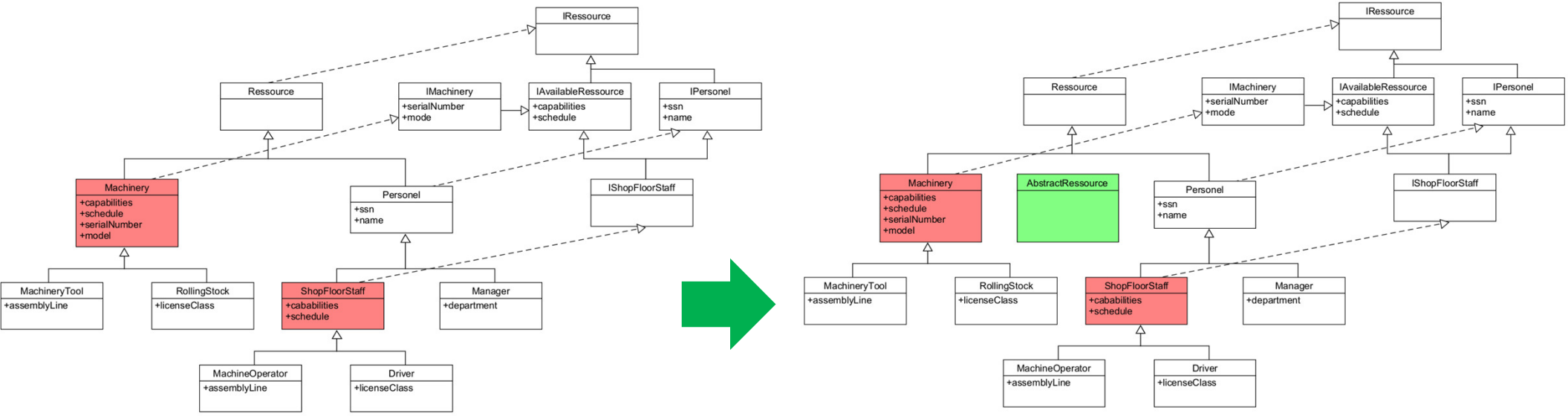
Algorithm

4. We create a new abstract class for reuse



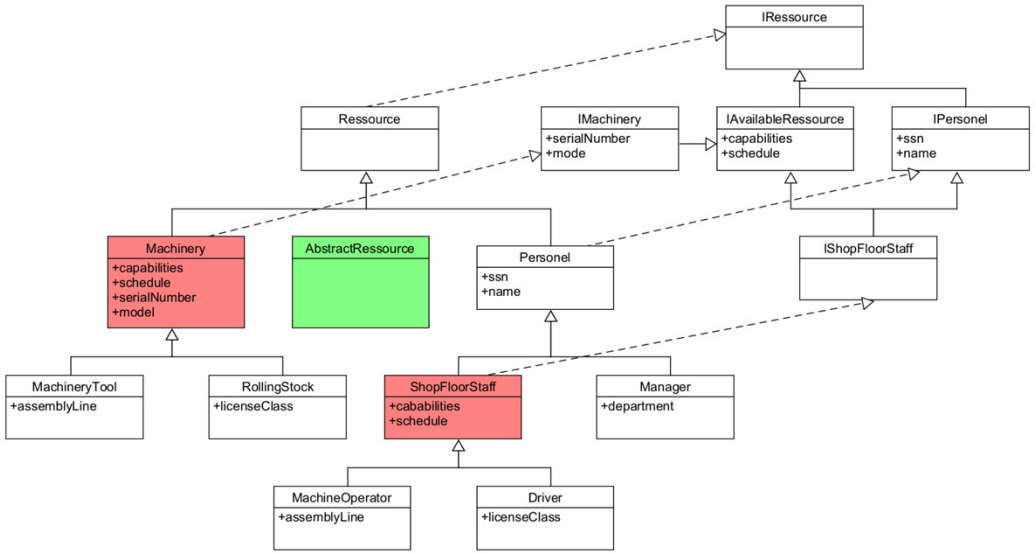
Algorithm

4. We create a new abstract class for reuse



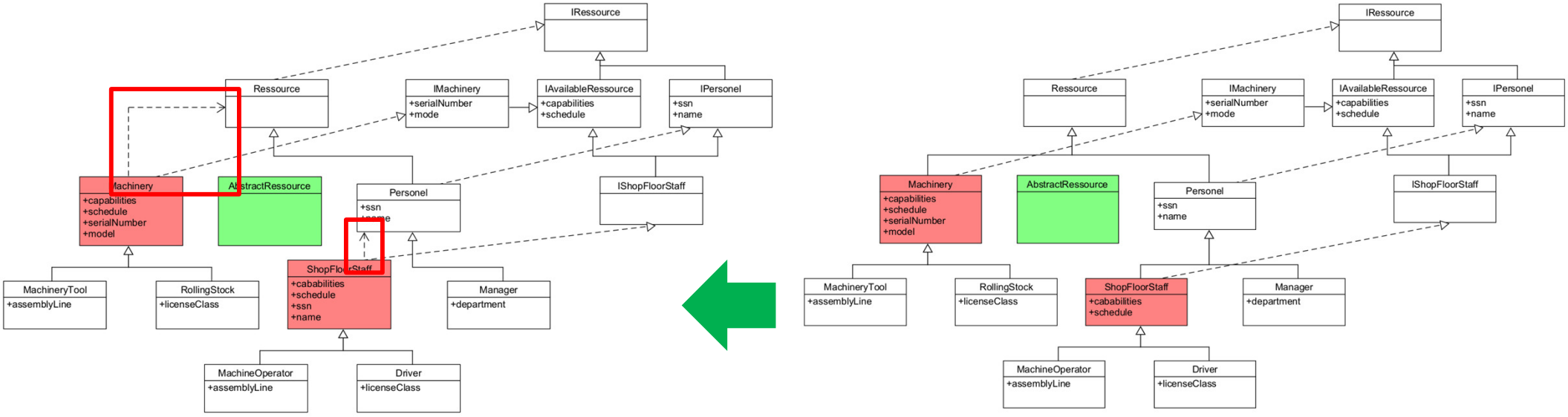
Algorithm

5. (We replace inheritance with delegation)



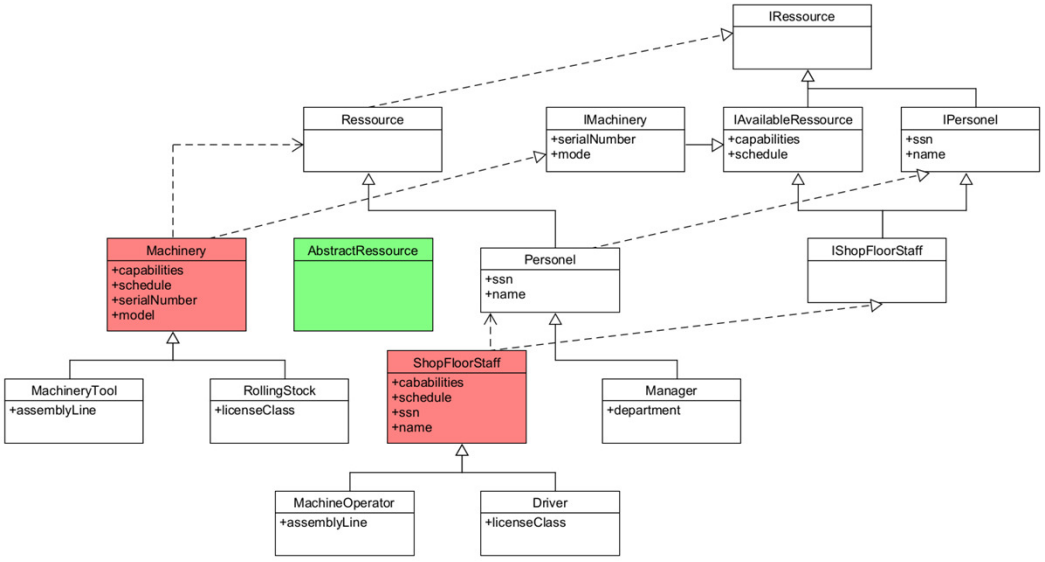
Algorithm

5. (We replace inheritance with delegation)



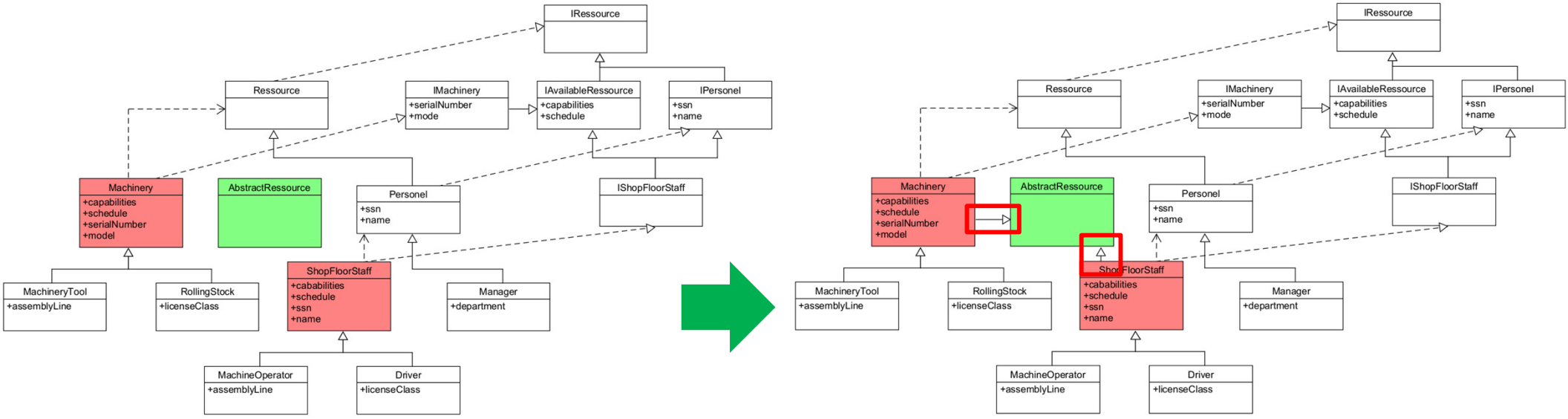
Algorithm

6. We make the new abstract class a superclass of the classes in the extent



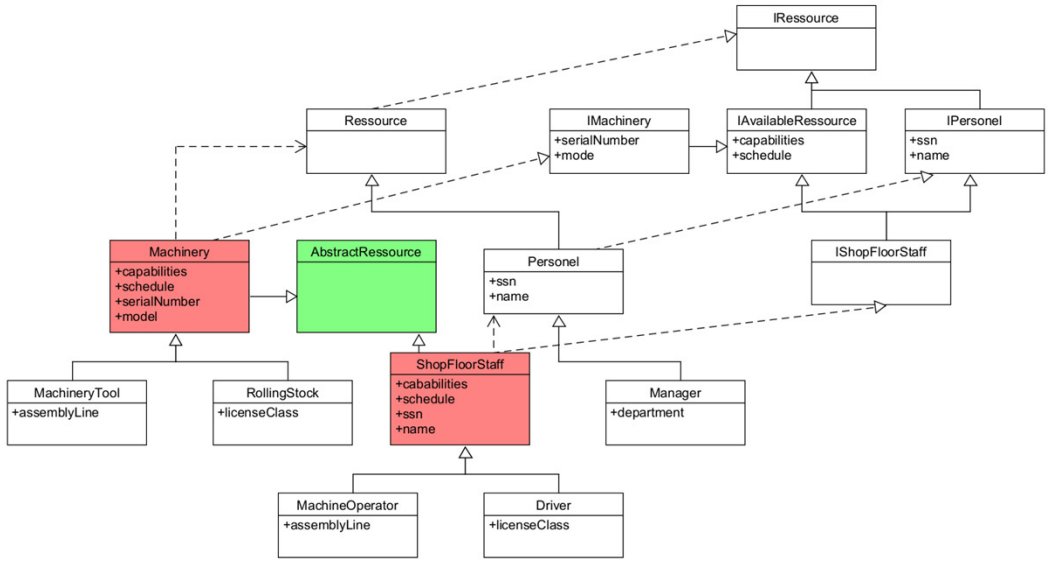
Algorithm

6. We make the new abstract class a superclass of the classes in the extent



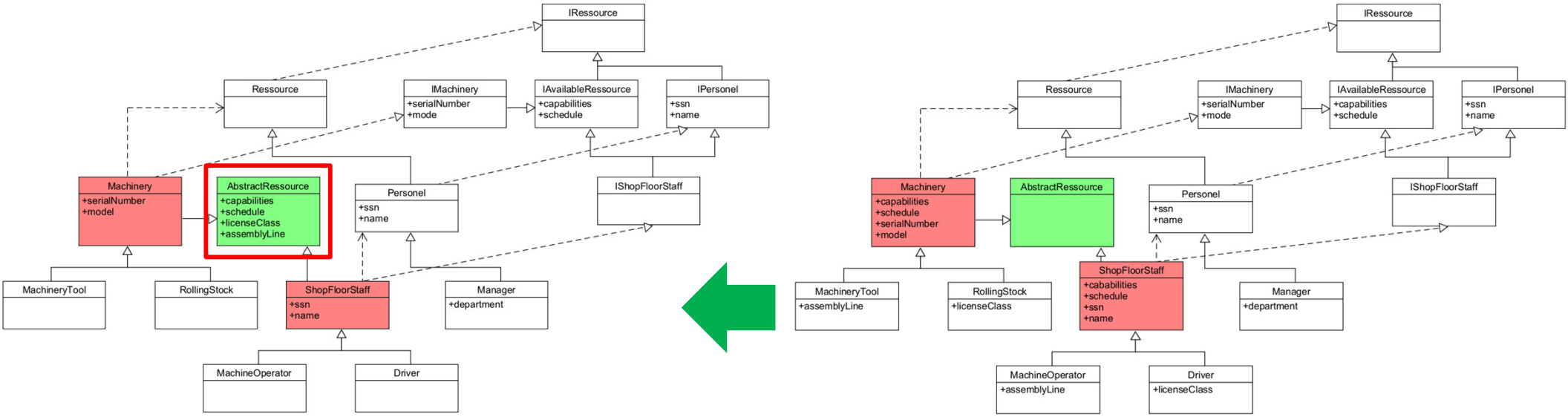
Algorithm

7. We pull up the method in the intent into the abstract class



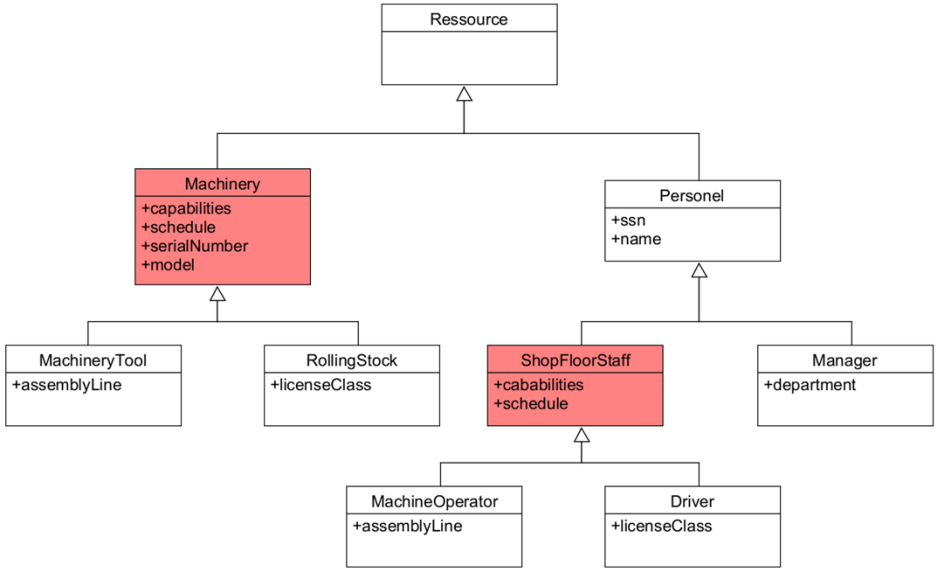
Algorithm

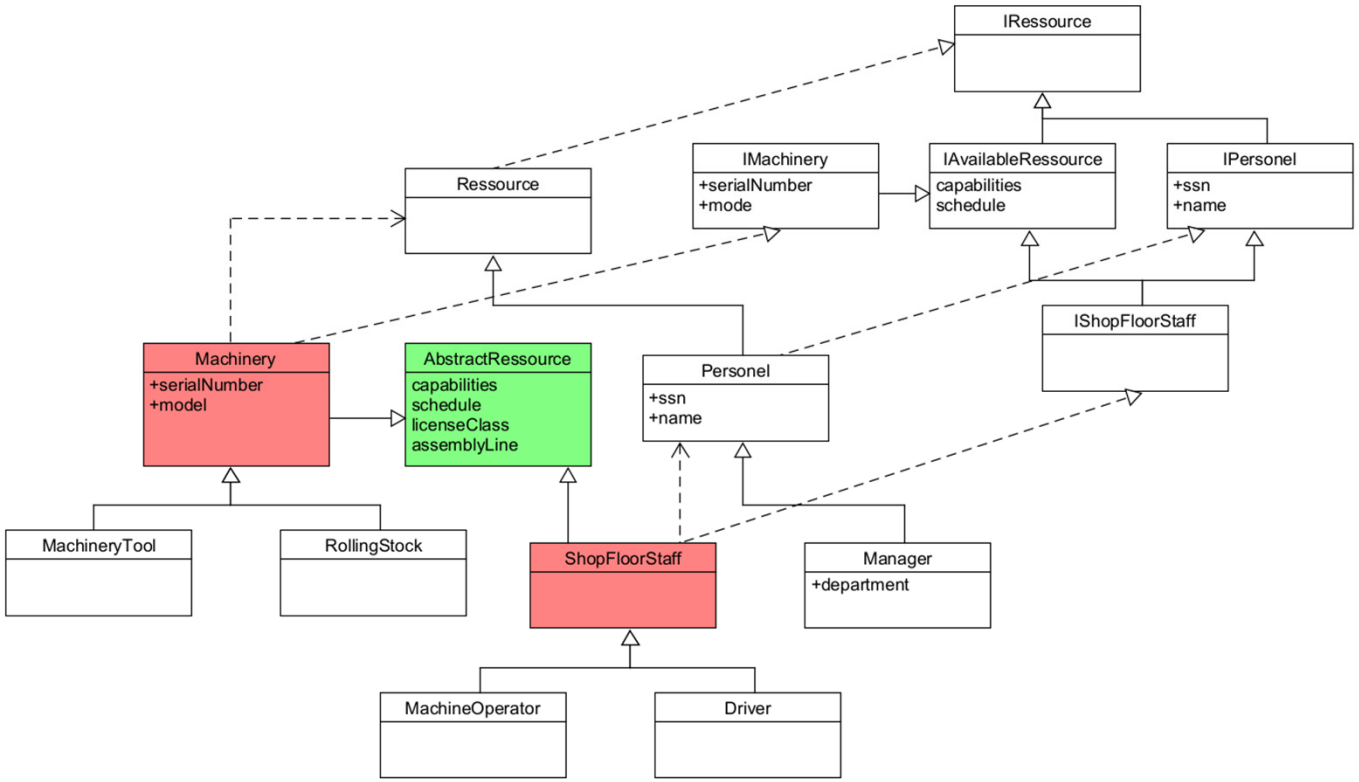
7. We pull up the method in the intent into the abstract class



Algorithm

- Summary

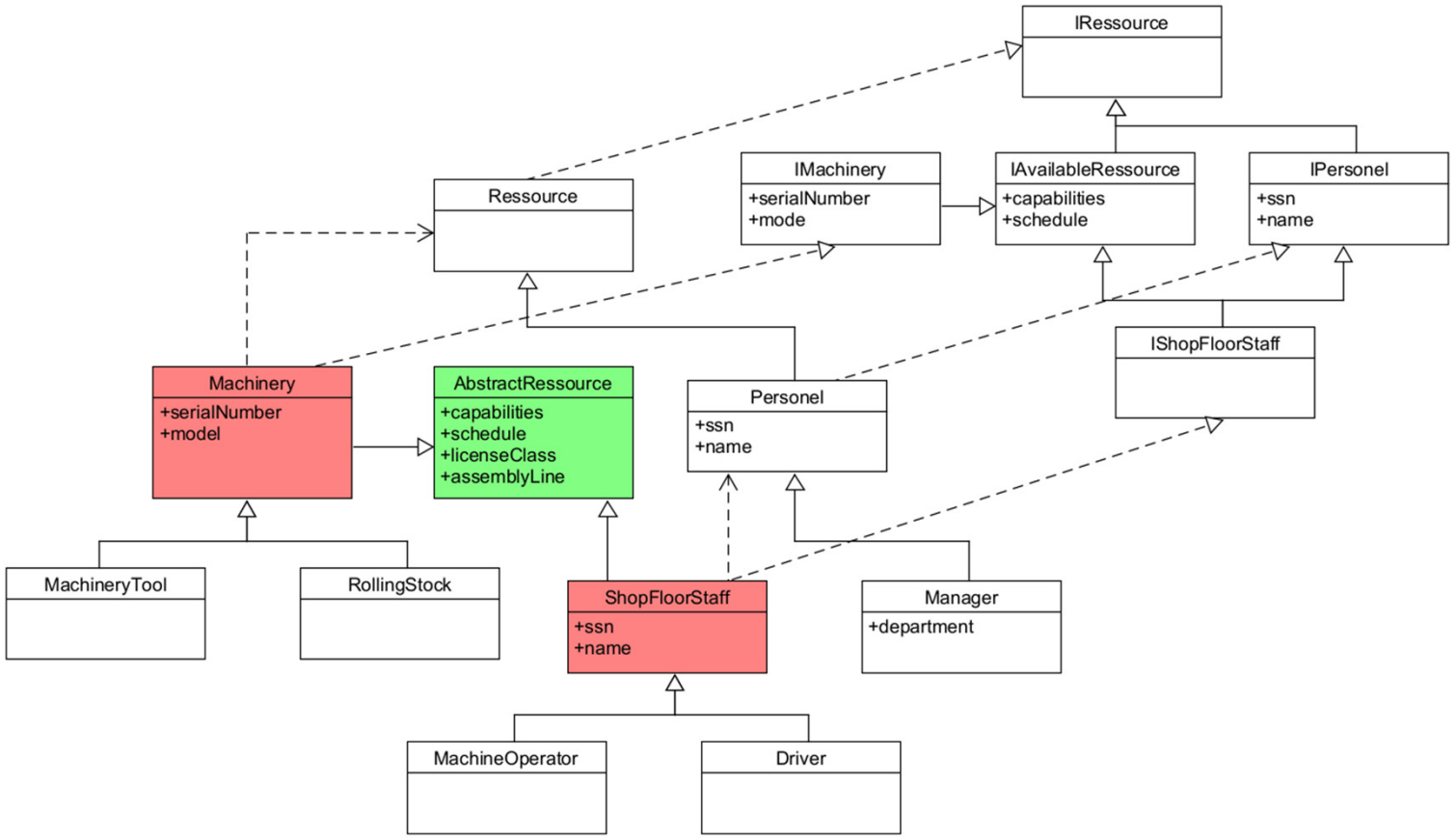




Limitations

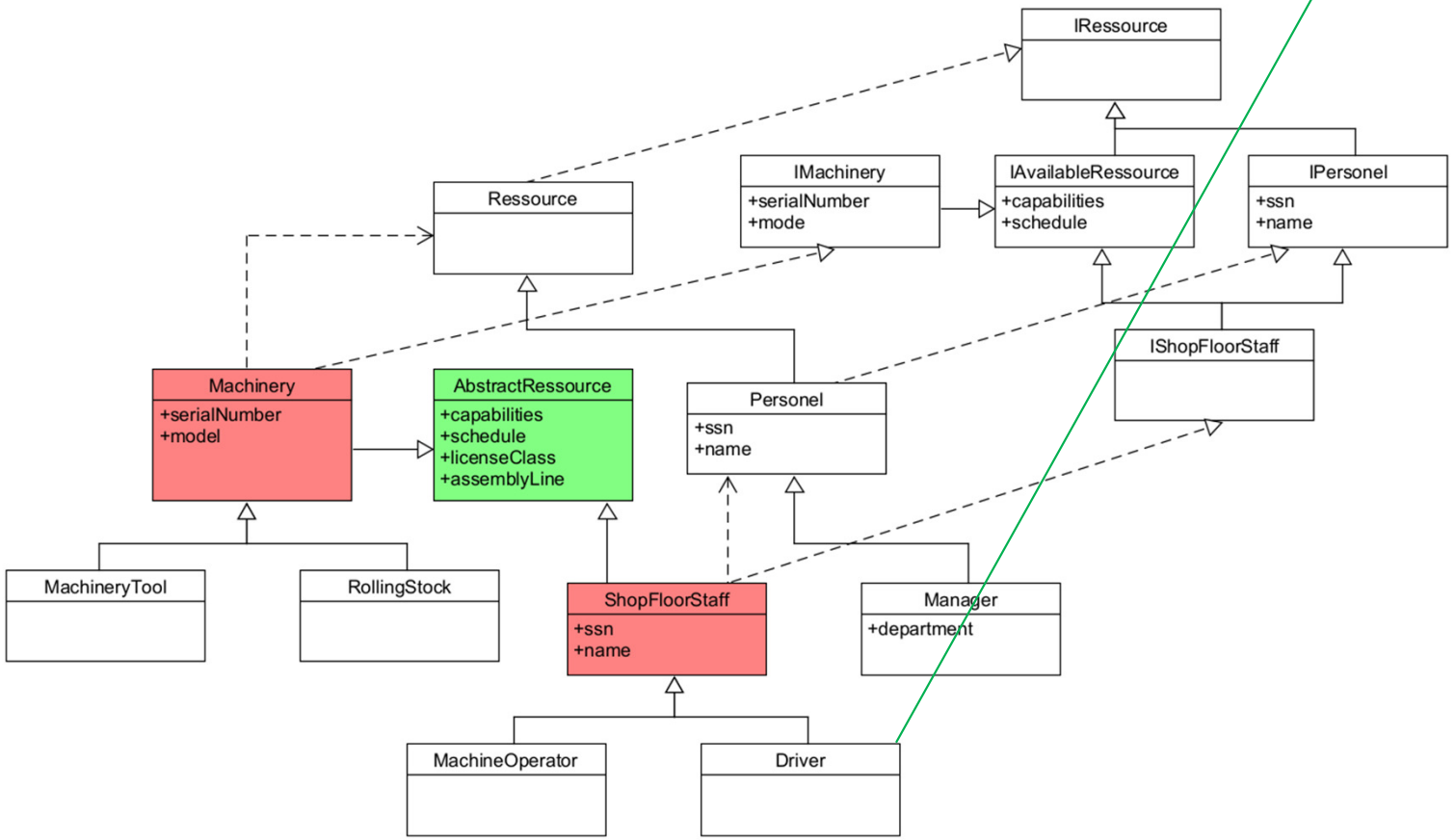
- Requires many atomic refactorings, rare cases where all the preconditions are fulfilled
  - Manual adjustments
- Requires complete separation of typing and reuse hierarchies
  - Is that such a good idea for developers?
- Works for Java and its interfaces/classes
  - Generalisation to other languages?

Discussions

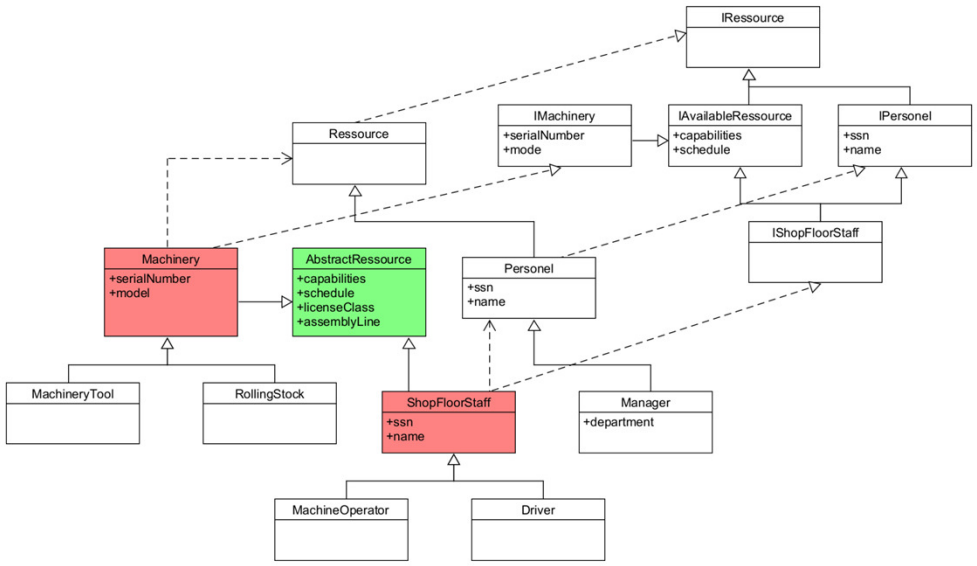


Discussions

Each subclass may receive extra methods



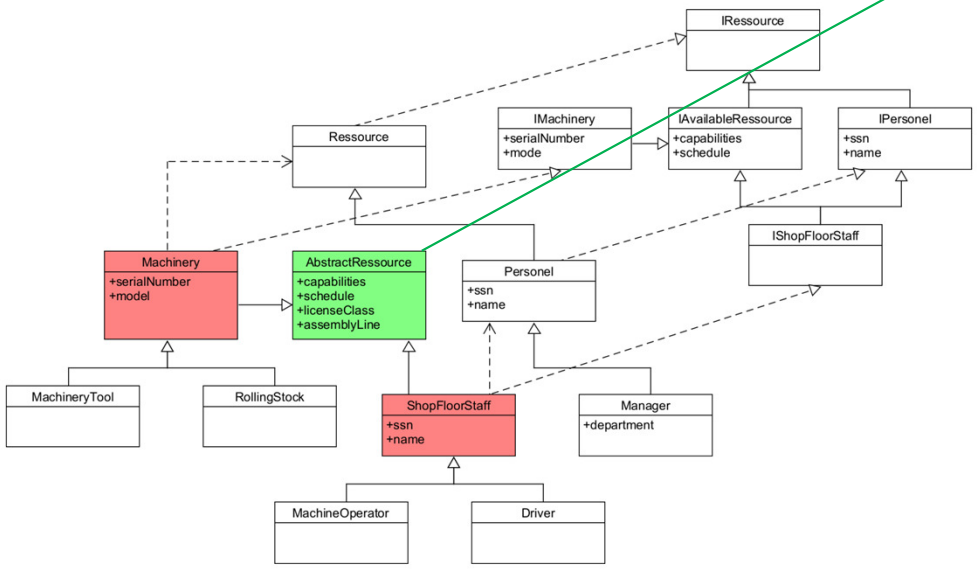
Discussions



- One solution could be to extend Steps 2 and 3 to the subclasses
  - Often many subclasses
  - The lattice built via FCA would then grow a lot
  - In this simple example, we already get 11 interfaces in total with the subclasses

Discussions

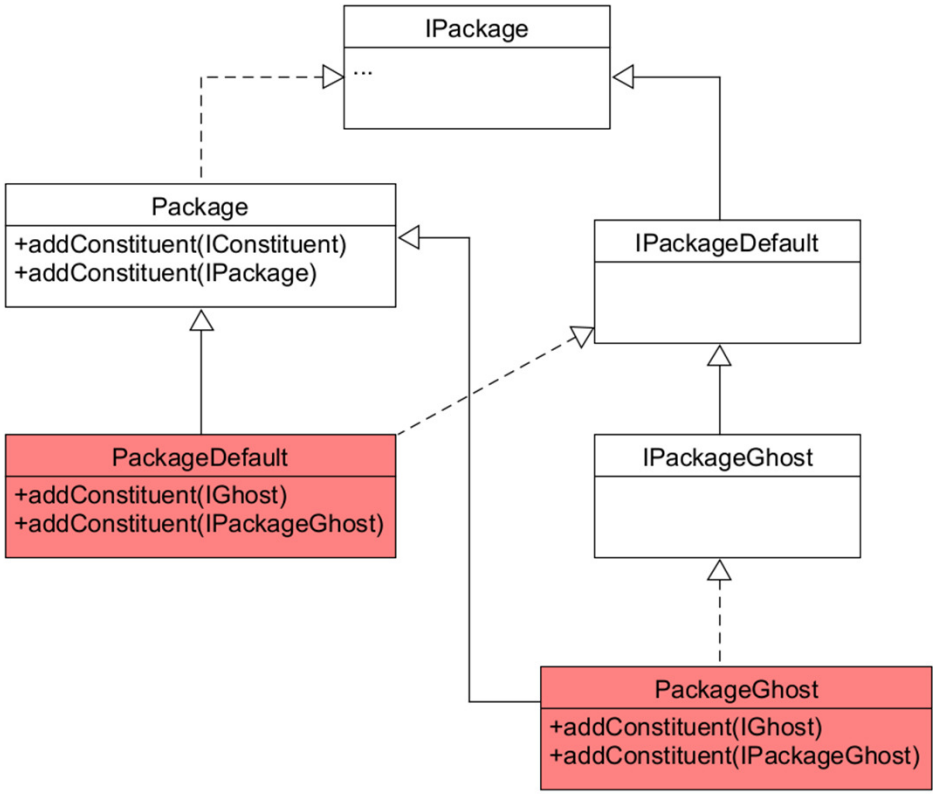
The visibility of the methods could be set to `protected` and then be increased later



- One solution could be to extend Steps 2 and 3 to the subclasses
  - Often many subclasses
  - The lattice built via FCA would then grow a lot
  - In this simple example, we already get 11 interfaces in total with the subclasses

Discussions

PADL

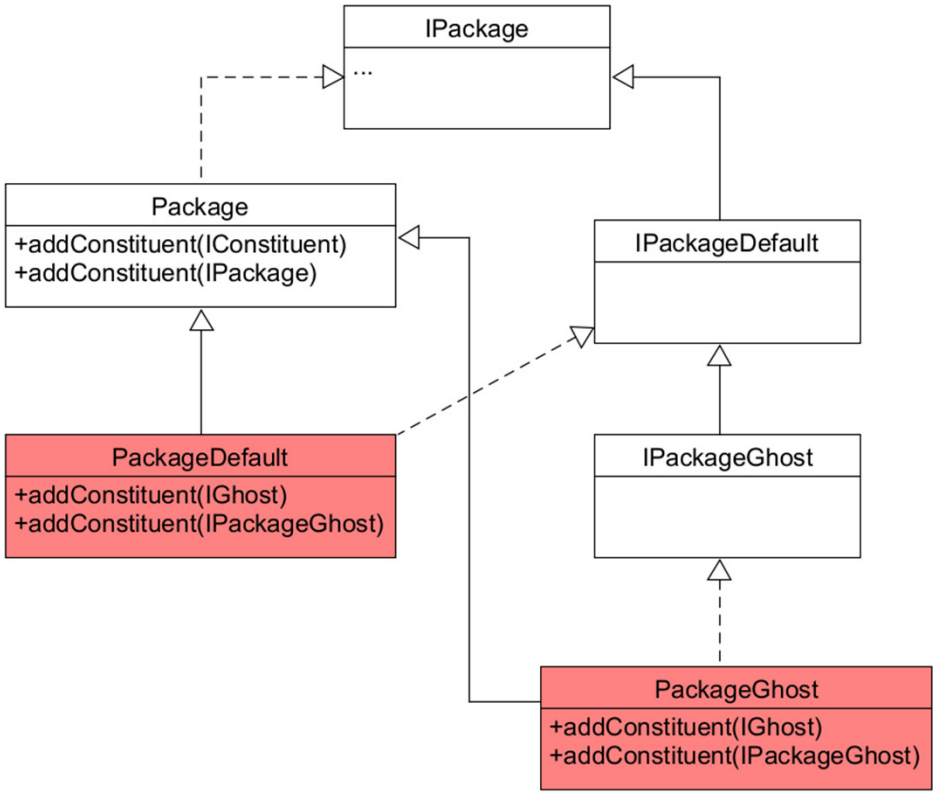


In `padl.kernel` and `….impl`

- In some cases, we can find better refactorings
  - An `IGhost` is an `IConstituent` and an `IPackageGhost` is an `IPackage`
  - We can delete the duplicated methods from `PackageDefault` and `PackageGhost`

Discussions

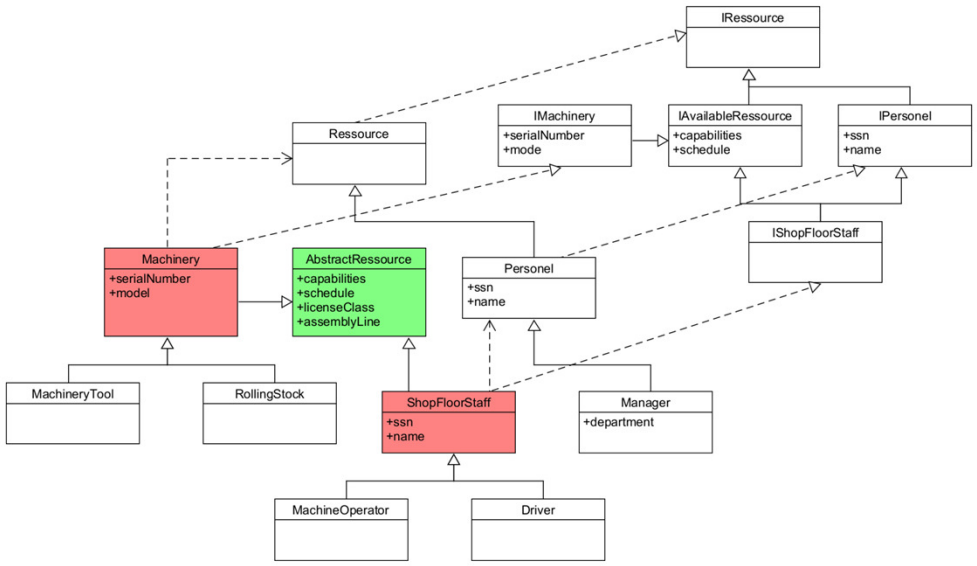
PADL



In `padl.kernel` and `….impl`

- However, it is hard to detect/handle all cases
  - GLASS could be used semi-automatically
  - Developers could look at a feature first, and then decide which refactorings to apply

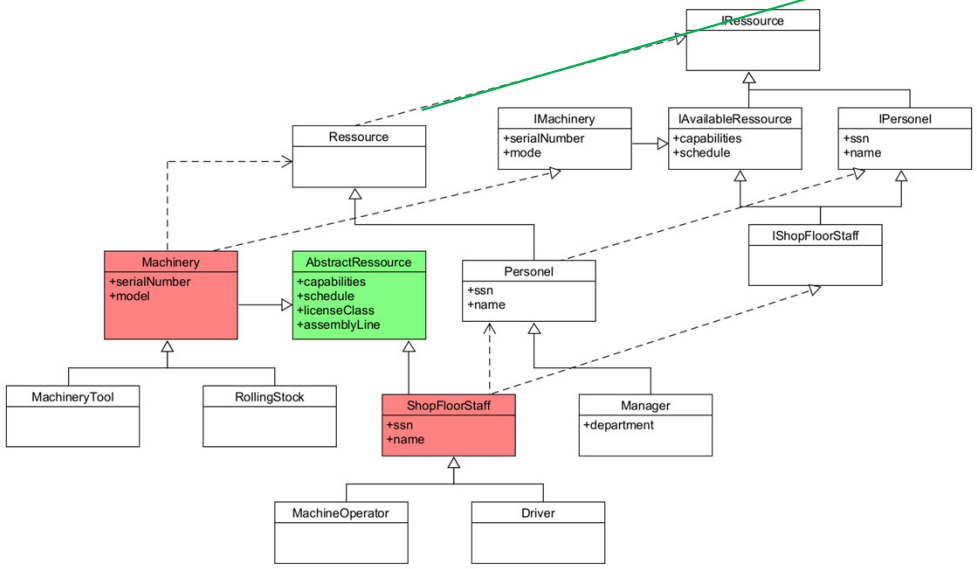
Discussions



- There could also be cases where it is more convenient to keep the hierarchy and delegate to the new class

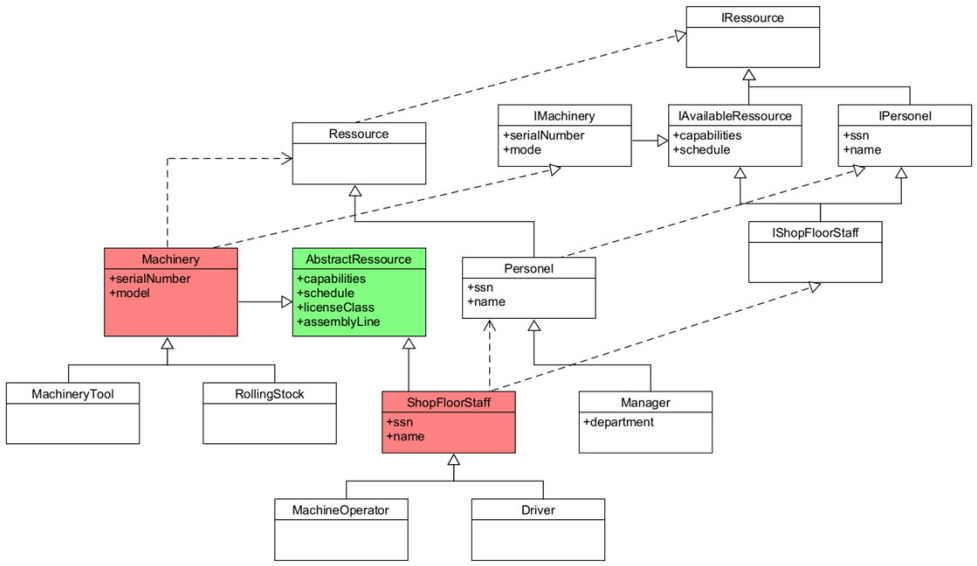
Discussions

If `Ressource` contained many methods, reuse may improve if kept as the superclass of `Machinery`



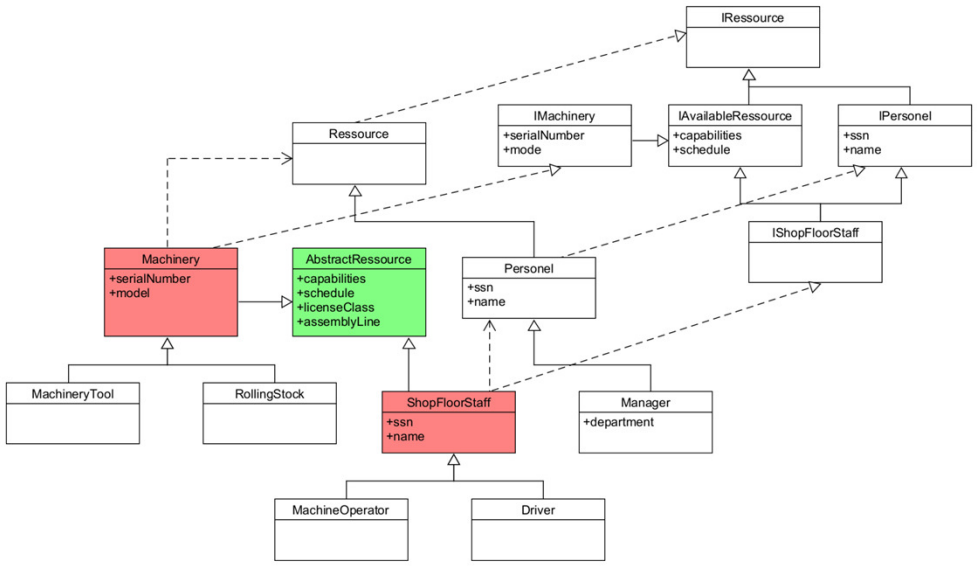
- There could also be cases where it is more convenient to keep the hierarchy and delegate to the new class

Discussions



- If there are different implementations, it is also not obvious to choose which one to pull up to the new class

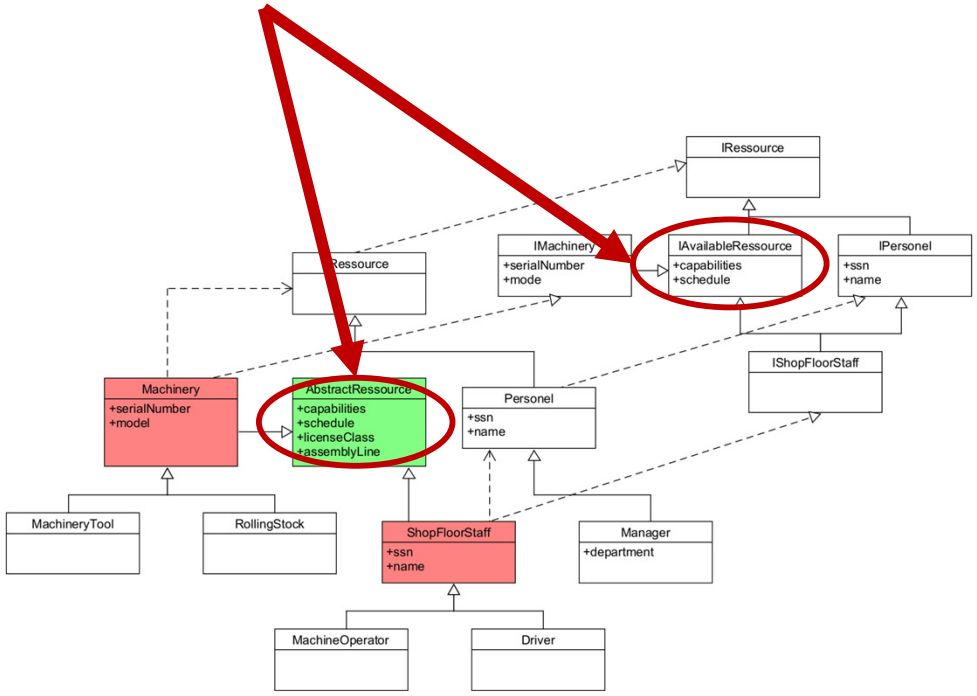
Discussions



- After applying this refactoring, new ad hoc features
  - Such feature could be called “Abstract interface reuse”
    - (Name is still undecided)

Discussions

Two independent occurrences of {`capabilities`, `schedule`}



- After applying this refactoring, new ad hoc features
  - Such feature could be called “Abstract interface reuse”
    - (Name is still undecided)

Discussions

- Does the introduction of our refactoring really improve code quality?
  - Ad hoc features may group together classes that are in different packages
    - Is it worth it to introduce dependencies between them?
  - A method that is duplicated may have different implementations, thus the refactoring will not reduce the amount of code
    - However, it might make it easier to extend the software in the future, as we have introduced classes/interfaces that facilitate code reuse

Discussions

Some methods will need to be redefined multiple times

- Does the introduction of our refactoring really improve code quality?
  - Ad hoc features may group together classes that are in different packages
    - Is it worth it to introduce dependencies between them?
  - A method that is duplicated may have different implementations, thus the refactoring will not reduce the amount of code
    - However, it might make it easier to extend the software in the future, as we have introduced classes/interfaces that facilitate code reuse

CONCLUSION

Conclusion

- FCA-based approach to discover functional features in OO programs
  - Including “missed” features
- FCA-based approach to suggest refactorings
  - Separate types from inheritance
  - Reduce code duplication

Conclusion

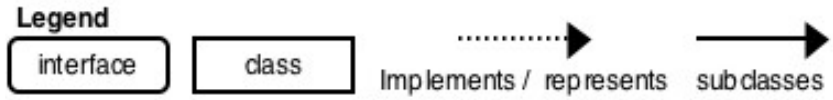
- Algorithmic, reproducible approaches
  - No LLMs were harmed during this research
  - “Vibe coding” makes such approaches even more relevant and necessary

Future Work

- Define and use quality models
  - Measure relevant characteristics
  - Assess trade-offs
    - Code duplication vs. Extra interfaces/classes

Future Work

- Automate the feature-refactoring approach
  - Can we use the type of the parent feature to decide if a refactoring is necessary?
- Automate the naming of the new interfaces and classes
  - Appropriate use of LLMs



Main Claim

- Typing hierarchy
- Reuse hierarchy