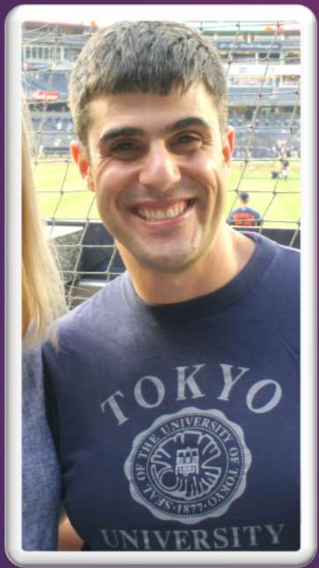
Automated Refactoring of Legacy Java Software to Default Methods

RAFFI KHATCHADOURIAN
CITY UNIVERSITY OF NEW YORK

HIDEHIKO MASUHARA
TOKYO INSTITUTE OF TECHNOLOGY



INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2017

Background: Default Methods in Java 8

- methods with bodies in interfaces
- introduced in Java 8
- useful to improve skeletal implementations

Long Awaited Java 9.0 Releasing This Week

Like | by Amit K Gupta on Sep 18, 2017. Estimated reading time: 5 minutes | Discuss

Share

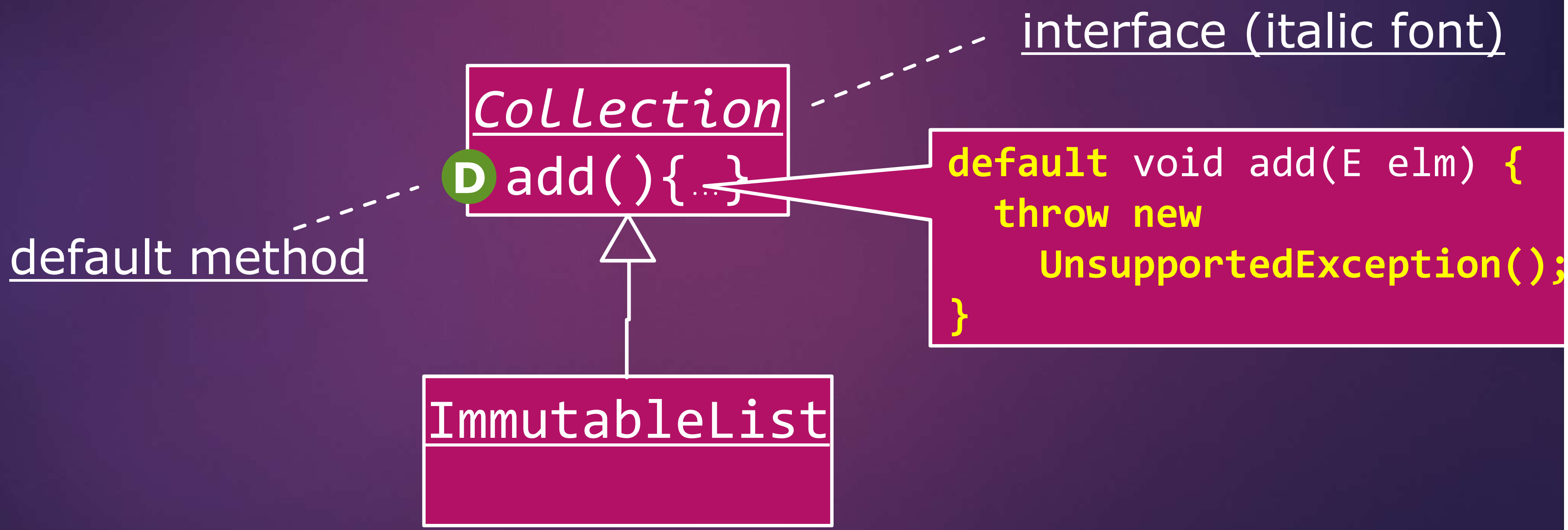
The long awaited next release of Java SE, version 9, is being released on September 21, 2017 and with it come some major changes.

The key change in JDK 9 is the introduction of "a new kind of Java programming component, th module, which is a named, self-describing collection of code and data", according to Oracle. T key aim of the modules technology is to reduce the size and complexity of both Java applicatio and the core Java runtime itself. To this end, the JDK itself has been modularized, an approac that Oracle intend to improve performance, security and maintainability.

To support Java 9 modules, a new modular JAR file with a module-info.class file in its root directory has also been introduced. Oracle have also introduced tooling to allow a set of mo to be assembled and optimized into a custom runtime image, without requiring a full Java ru to be present. Other changes as a consequence of modularisation include the removal of

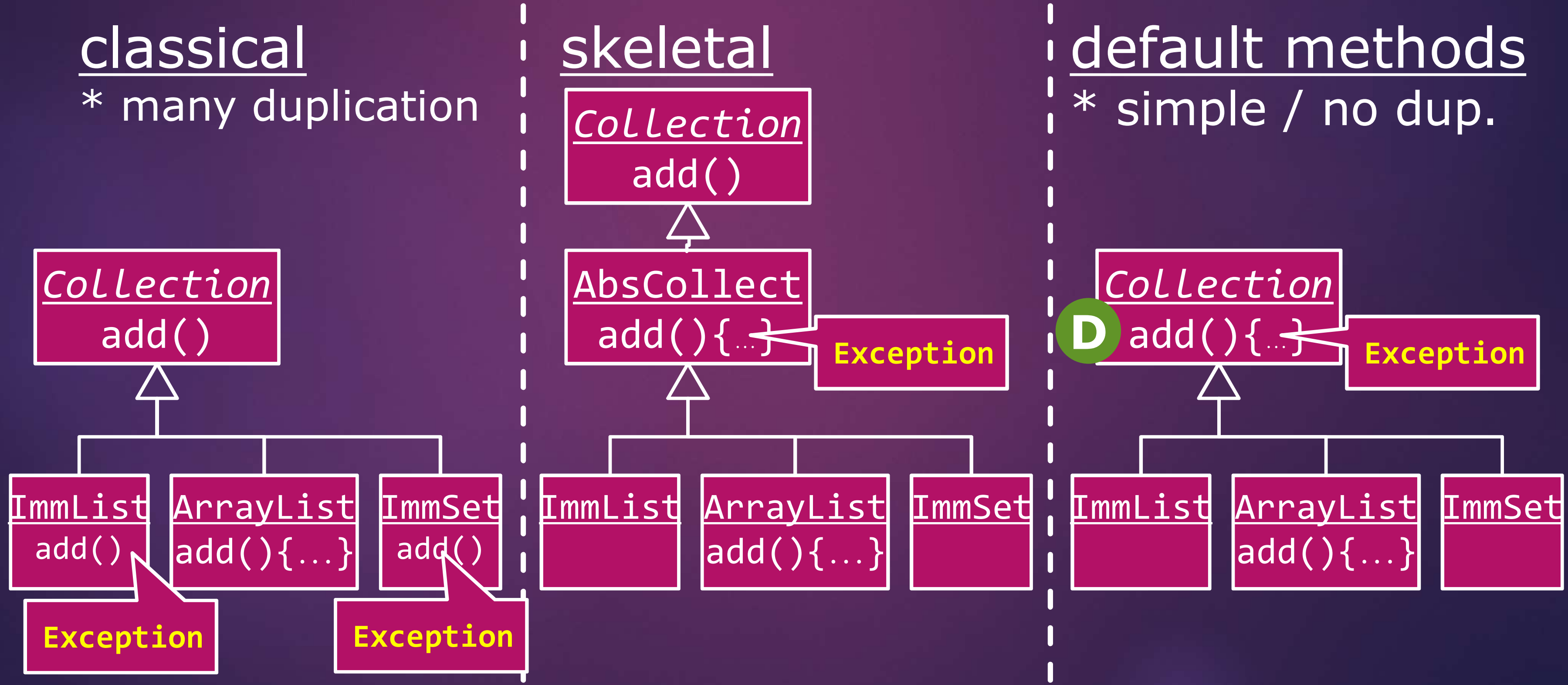
Background: default methods

- methods with bodies in interfaces
- (originally for interface evolution)



Background: usefulness of default methods

- alternative to skeletal impl. [Goetz, 2011]

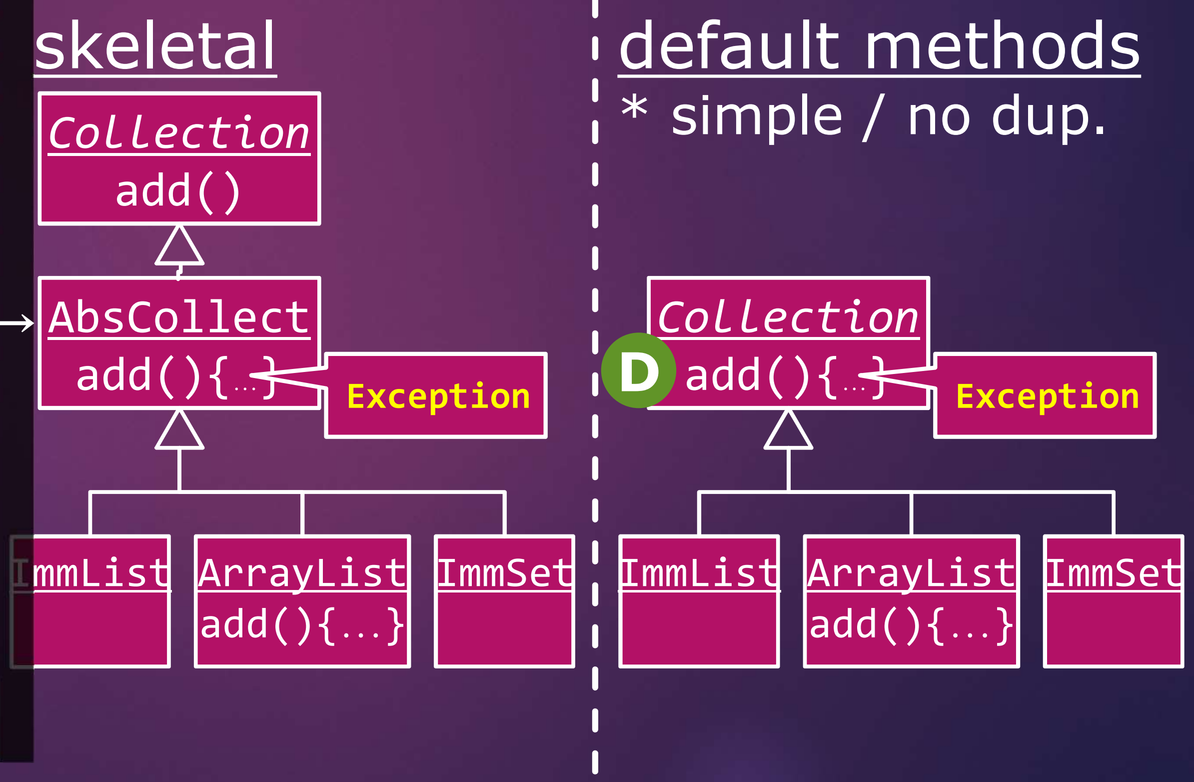


Background: usefulness of default methods

- alternative to skeletal impl. [Goetz, 2011]

Problems

- Inheritance: single tree only
- Modularity: need to find this→
- Bloat: +1 class



Problem: Migration can be Difficult

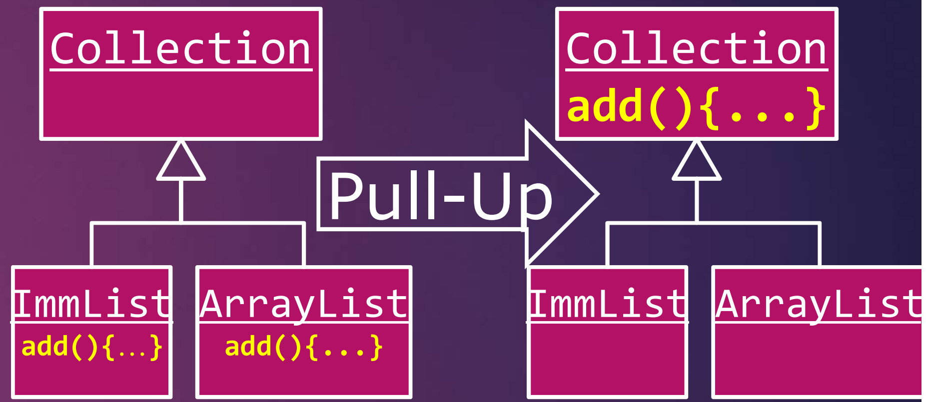
requiring significant manual effort because

- ubiquitous
- subtle semantic restrictions
  - type-correctness
  - multiple inheritance
  - diff. between class and interface
  - tie-breakers

Related: Pull-Up Method Refactoring?

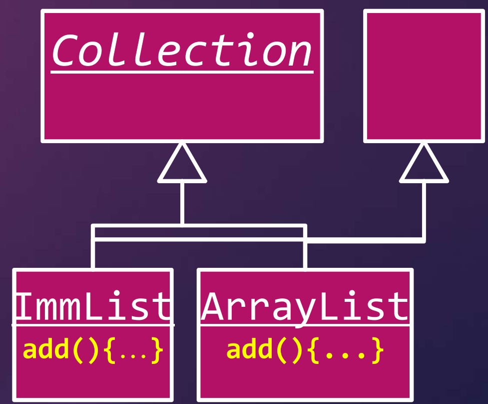
[Fowler99, Tip+11]

- moves methods from a subclass into a super class
  - for reducing redundancy



— Not directly.
as it is interfaces

  - multiple inheritance
  - “competition” with classes (tie-breaking)



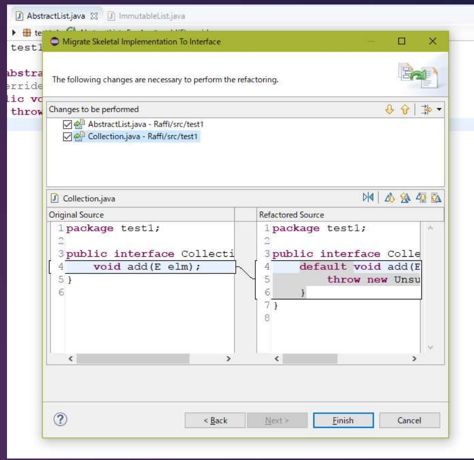
Related: "Move Original Method to Super Class"? [Borba+04]

- is a law expresses transformational semantic equivalence

— Not for method bodies.

- In our case, no method declarations are being moved but rather bodies

Contributions: a Refactoring Tool

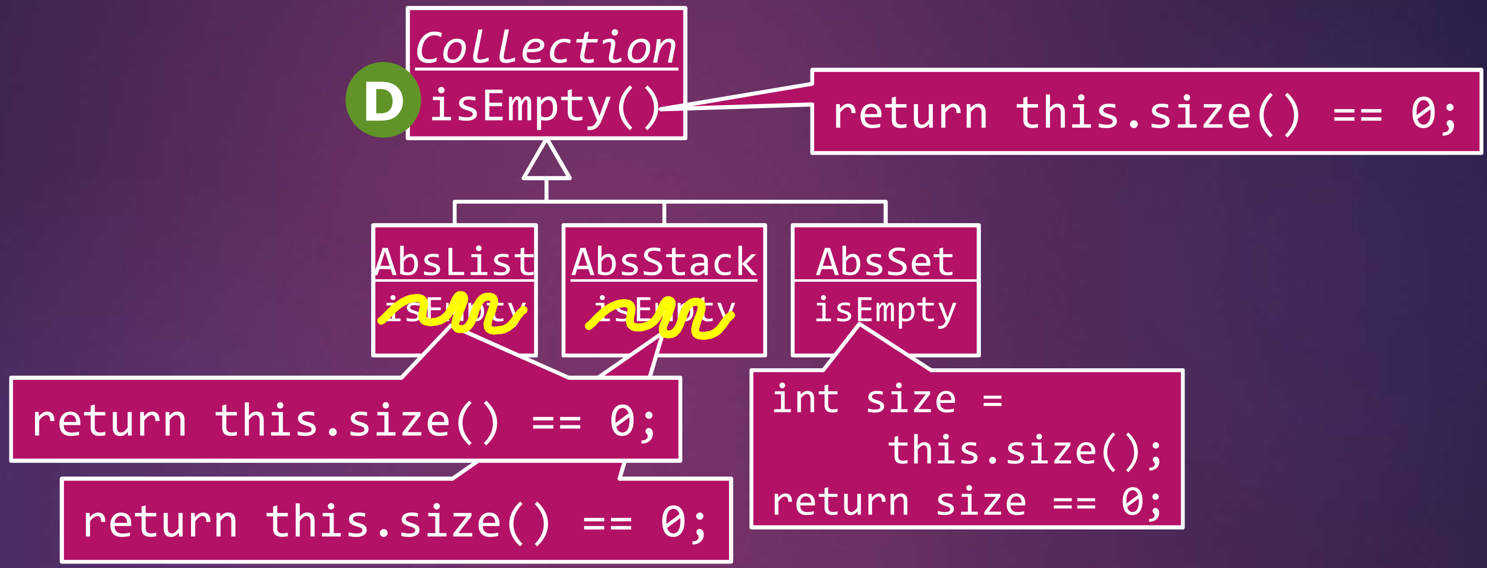


- developed a refactoring tool
  - as an Eclipse plugin
  - migrates into default methods
  - conservative; preserves semantics
- tested with open-source projects
  - to count successful/failed cases by applying the tool
  - to inquire developers' opinions by sending pull-requests

Approach

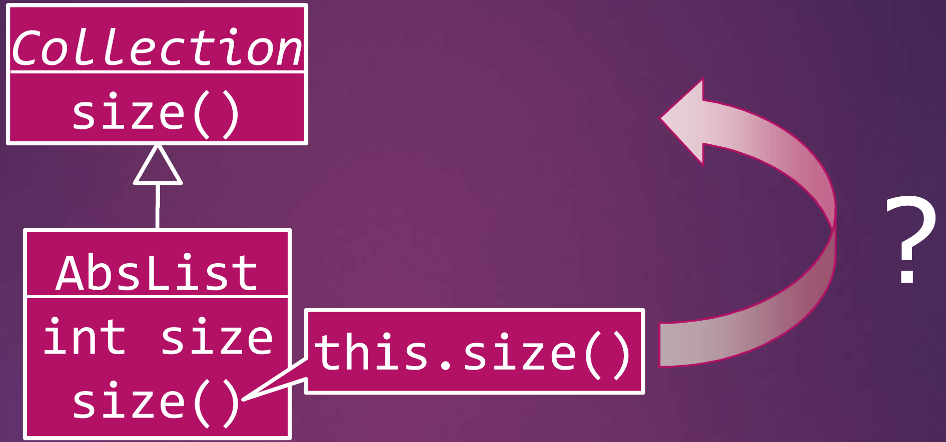
- For each candidate method and target interface
  - move the method
  - check preconditions
    for type-safety
    and semantic preservation
  - remove the methods with the same body in sibling classes

Contributions: Target Methods with Multiple Source Methods



- Safe to migrate any of them
- Which one to migrate?
- Choose the largest number of "equivalent" source methods

Interfaces cannot Declare Instance Fields



Q: In general, how can we guarantee that migration results in a type-correct transformation?

[Palsberg&Schwartzbach94,Tip+11]

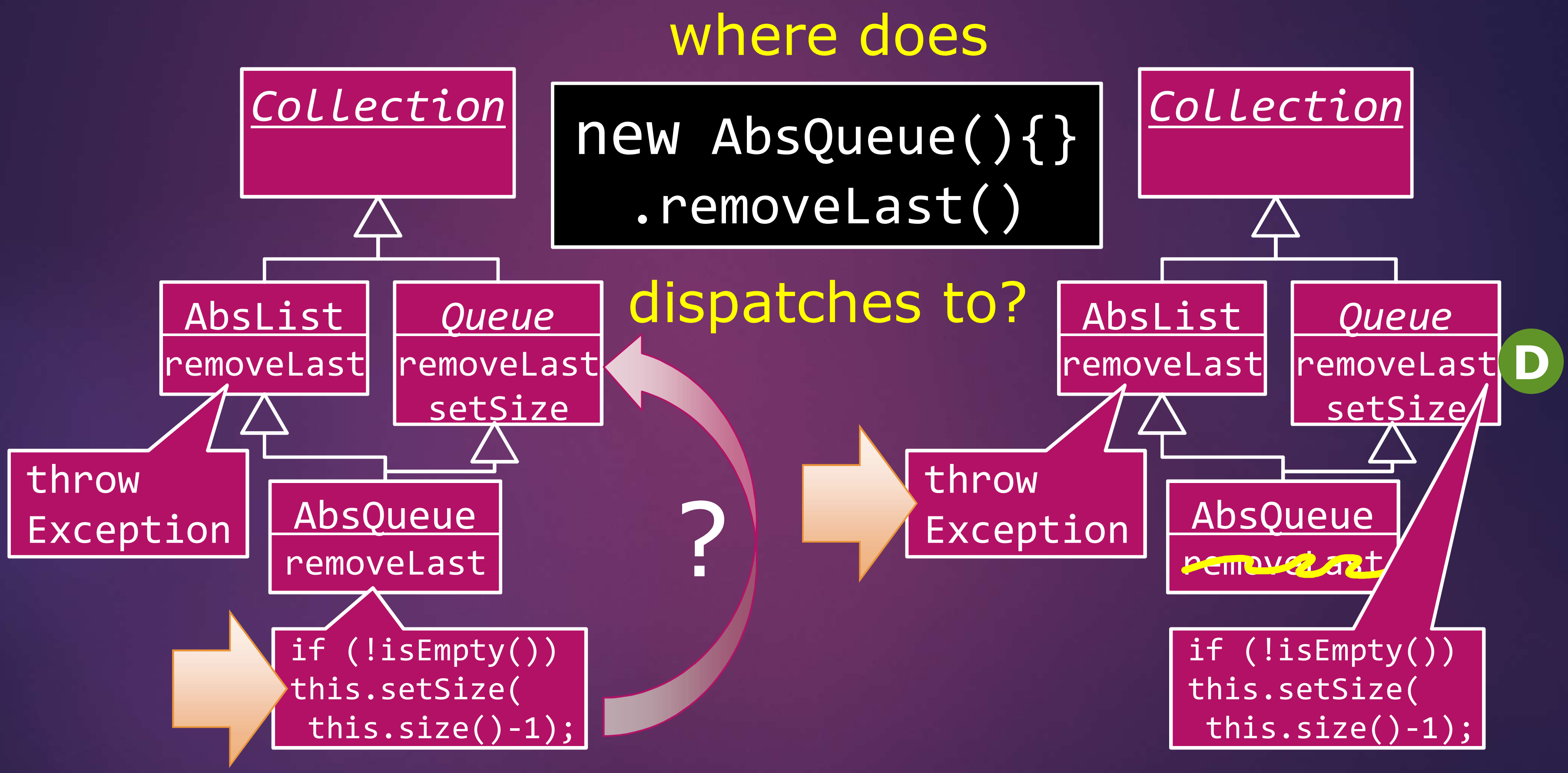
A: Use type constraints to check refactoring preconditions.

Preconditions for safety & semantic preconditions

program construct	implied type constraint(s)	
assignment $E_i = E_j$	$[E_j] \leq [E_i]$	(1)
method call $E.m(E_1, \ldots, E_n)$ to a virtual method M (throwing exceptions $Ex_{t1}, \ldots, Ex_{tj}$)	$[E.m(E_1, \ldots, E_n)] \triangleq [M]$	(2)
	$[E_i] \leq [Param(M, i)]$	(3)
	$[E] \leq Decl(M_1) \vee \cdots \vee [E] \leq Decl(M_k)$ where $RootDefs(M) = \{M_1, \ldots, M_k\}$	(4)
	$\forall Ex_t \in \{Ex_{t1}, \ldots, Ex_{tj}\}$ $\exists Ex_h \in Handle(E.m(E_1, \ldots, E_n))[[Ex_t] \leq [Ex_h]]$	(5)
access $E.f$ to field F	$[E.f] \triangleq [F]$	(6)
	$[E] \leq Decl(F)$	(7)
return E in method M	$[E] \leq [M]$	(8)
M' overrides M, $M' \neq M$	$[Param(M', i)] = [Param(M, i)]$	(9)
	$[M'] \leq [M]$	(10)
	$Decl(M') < Decl(M)$	(11)
for every class (and enum) C	$C \leq$ `java.lang.Object`	(12)
for every interface I	$I \not\leq$ `java.lang.Object` $\wedge \forall M[Decl(M) \triangleq$ `java.lang.Object` $\wedge Public(M) \implies \exists M'[Decl(M') \triangleq I \wedge NOverrides(M', M)]]$	(13)
for every functional interface I	$\exists M[Decl(M) \triangleq I \wedge Abstract(M) \wedge \forall M'[Decl(M') \triangleq I \wedge M' \neq M \implies \neg Abstract(M')]]$	(14)
implicit declaration of **this** in method M	$[$**this**$] \triangleq Decl(M)$	(15)
implicit declaration of **super** in method M	$\neg Interface(Decl(M)) \implies [$**super**$] \triangleq super(Decl(M))$	(16)
implicit declaration of `I.`**super** in method M	$Decl(M) < I \implies [$`I.`**super**$] \triangleq I$	(17)
expression **new** $T(E_1, \ldots, E_n) \ldots$	$[$**new** $T(E_1, \ldots, E_n) \ldots] \leq [T]$	(18)
declaration of method M (declared in type T)	$Decl(M) \triangleq T$	(19)
declaration of field F (declared in type T)	$Decl(F) \triangleq T$	(20)
explicit declaration of variable or method parameter $T\ v$	$[v] \triangleq T$	(21)
declaration of method M with return type T	$[M] \triangleq T$	(22)
declaration of field F with type T	$[F] \triangleq T$	(23)
cast $(T)E$	$[(T)E] \triangleq T$	(24)
declaration of method M declared in interface I	$\exists J, M'[Interface(J) \wedge J \not\leq I \wedge I \not\leq J \wedge J \leq Decl(M') \wedge NOverrides(M', M) \wedge (Default(M') \vee Default(M))] \implies \forall C \mid Class(C) \wedge C < I \wedge C < J[\exists M''[M'' \neq M' \wedge M'' \neq M \wedge Class(Decl(M'')) \wedge C \leq Decl(M'') \wedge Public(M'') \wedge NOverrides(M'', M')]]$	(25)
declaration of concrete type T implementing interface I declaring method M	$\exists M'[T \leq Decl(M') \wedge NOverrides(M, M') \wedge \neg Abstract(M') \wedge \forall M''[T < Decl(M'') < Decl(M') \wedge NOverrides(M'', M') \implies \neg Abstract(M'')]]$	(26)

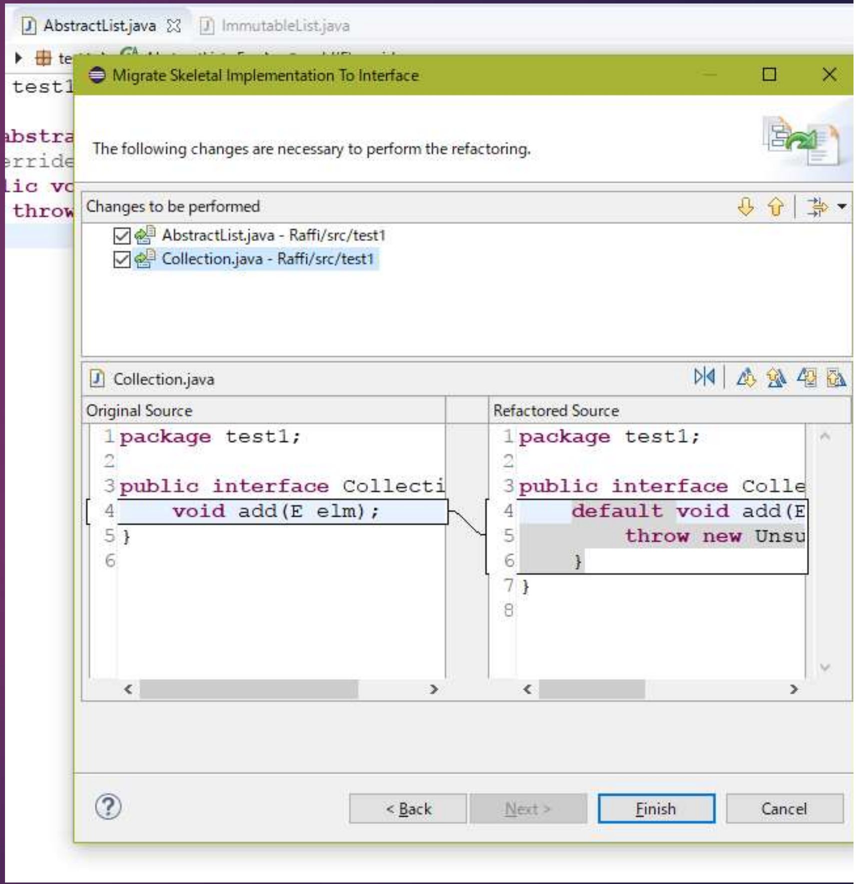
- Type safety rules + semantic preservation rules
- Extended from [Tip+11] for default methods
- See paper for more details

Preserving Semantics in Light of Multiple Inheritance



Eclipse Plug-in and Case Study

- Implemented as an Eclipse plug-in
- Applied to 19 Java programs
  - how many methods can be migrated?
  - efficient enough?
  - when methods cannot be migrated?



Eclipse Plug-in and Case Study (Result)

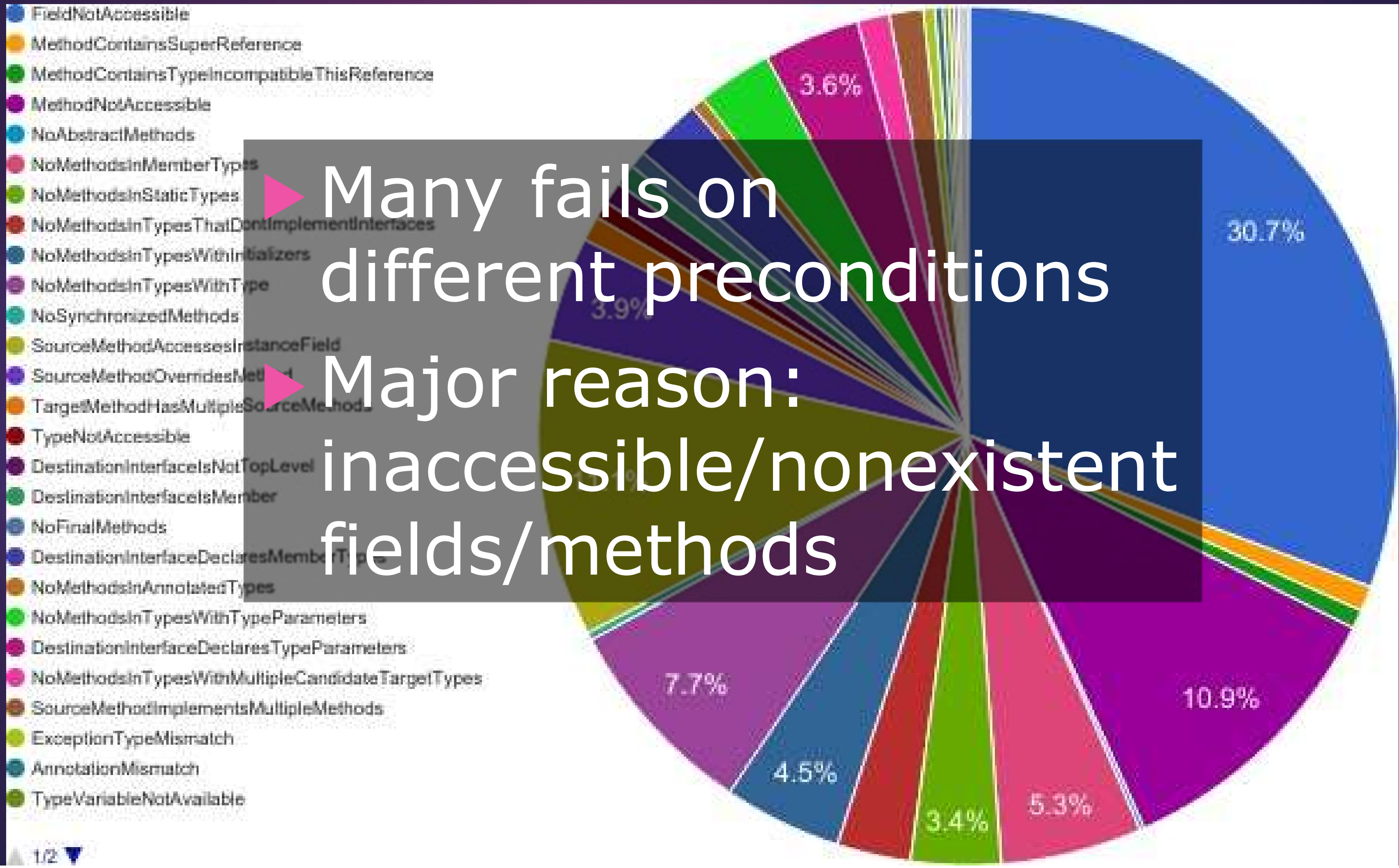
subject	KL	KM	cnds	dflts	fps	δ	-δ	tm (s)
ArtOfIllusion	118	6.94	16	1	34	1	0	3.65
Azureus	599	[illegible]	747	116	[illegible]	31	2	61.83
Colt	36	3.77	69	4	140	3	0	6.76
elasticsearch	585	47.87	339	69	644	21	4	83.30
Java8	291	[illegible]	209	93	775	25	10	64.66
JavaPush	6	0.77	1	0	4	0	0	1.02
JGraph	13	1.47	16	2	21	1	0	3.12
JHotDraw	32	3.60	181	46	282	8	[illegible]	7.75
JUnit	26	3.58	9	0	25	0	0	0.79
MWDumper	5	0.40	11	0	24	0	0	0.29
osgi	18	1.81	13	3	11	2	0	0.76
rdp4j	2	0.26	10	8	2	1	0	1.10
spring	506	53.51	776	150	1459	50	13	91.68
Tomcat	[illegible]	16.15	233	31	399	13	0	13.81
verbose	4	0.55	1	0	1	0	0	0.55
VietPad	11	[illegible]	16	0	26	0	0	0.36
Violet	27	2.06	104	40	102	5	1	3.54
Wezzle2D	35	2.18	87	13	181	5	0	4.26
ZKoss	185	15.95	394	76	684	0	0	33.95
Totals:	2677	232.2	3321	652	6180	166	30	383.17

18% (30/166) classes can be removed

7 KLOC/s runtime

Automatically migrated 19.6% candidates (652/3321 methods)

Refactoring Precondition Failure Distribution



- Many fails on different preconditions
- Major reason: inaccessible/nonexistent fields/methods

(Preliminary) Pull Request Study

Q: "Is it useful in practice?"

Procedure:

1. Choose GitHub projects
2. Apply refactorings
3. Send pull requests
4. Wait

Result:

- 19 pull requests
  - **4 merged**
  - 5 still open
  - 10 rejected
- Reasons of rejection:
  - no Java 8 yet
  - support older clients (Android)
  - fear of performance
  - ...

List of Projects in Pull Request Study

Merged

- JSilhouette
- Eclipse Collections
- Cyclops React
- Bootique

Still open

- QBit
- JGit
- Java8 Commons
- Koral
- Dari
- Binnavi

Rejected

- Blueocean
- JUnit
- RxJava
- ElasticSearch
- Guava
- Spring Framework
- jOOQ
- Java Design Patterns
- Jetty

A Thought: Evaluation Methods of New Language Features

Autopsy

Investigating GitHub repo's.

- state of the art
- scales
- <u>can see adopted cases only</u>

Proactive

Sending pull requests

- this work
- manual effort
- <u>can learn reasons of rejection</u>

Summary

- A refactoring approach from skeletal implementation to default methods
  - efficient, fully-automated, semantics-preserving
  - based on type constraints
  - implemented as an Eclipse IDE plug-in
- Evaluated
  - refactored 19.63% of methods in 19 projects
  - 4 pull requests merged into 19 projects