Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

To do before class

- Watch videos
- Read chapters 8 and 11 in the textbook
- Send questions and opinions through slack

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Design, implementation, and maintenance I

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Which tools do we have for designing a system?

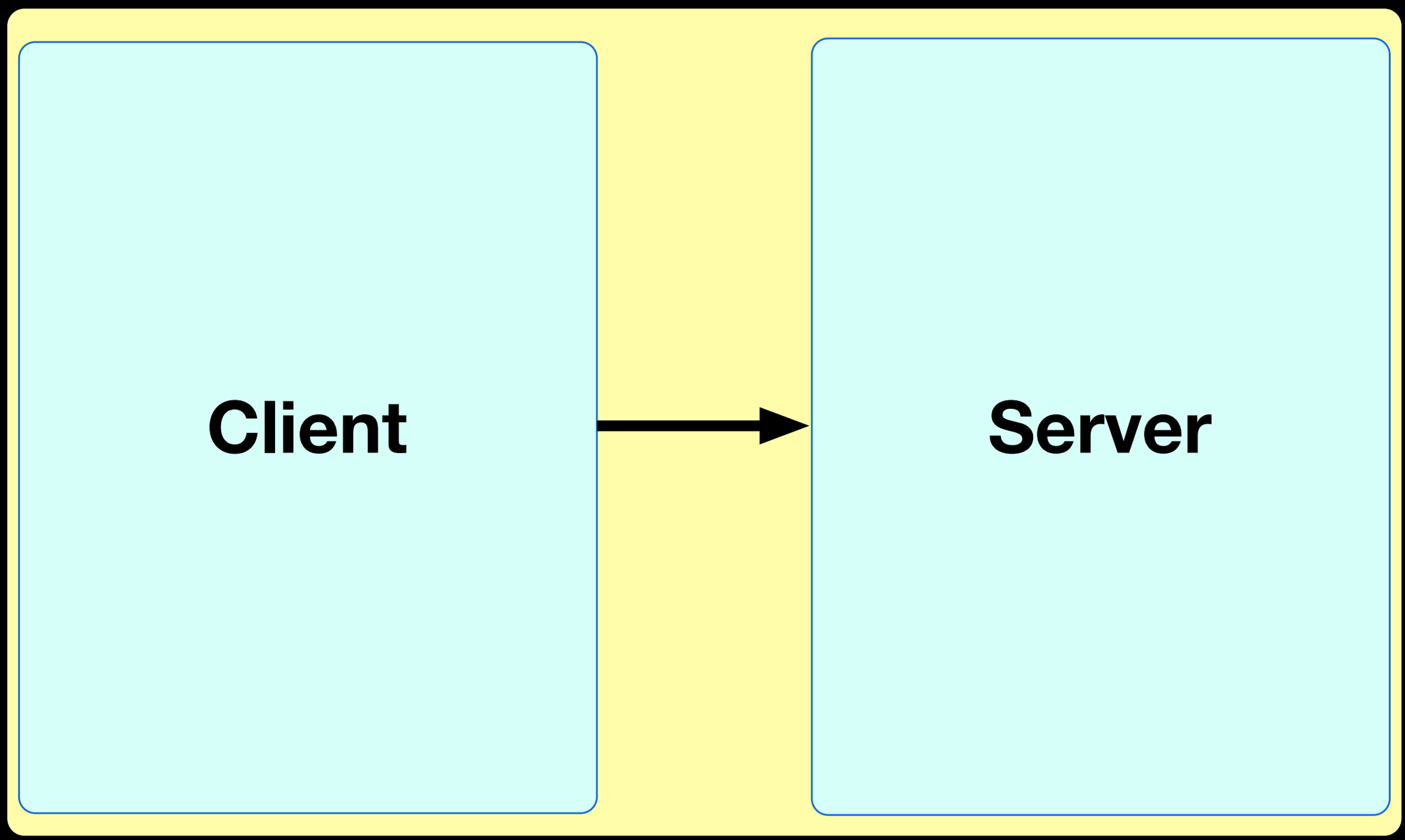
What are the main concepts and how do they relate?

How the system implementation will be structured?

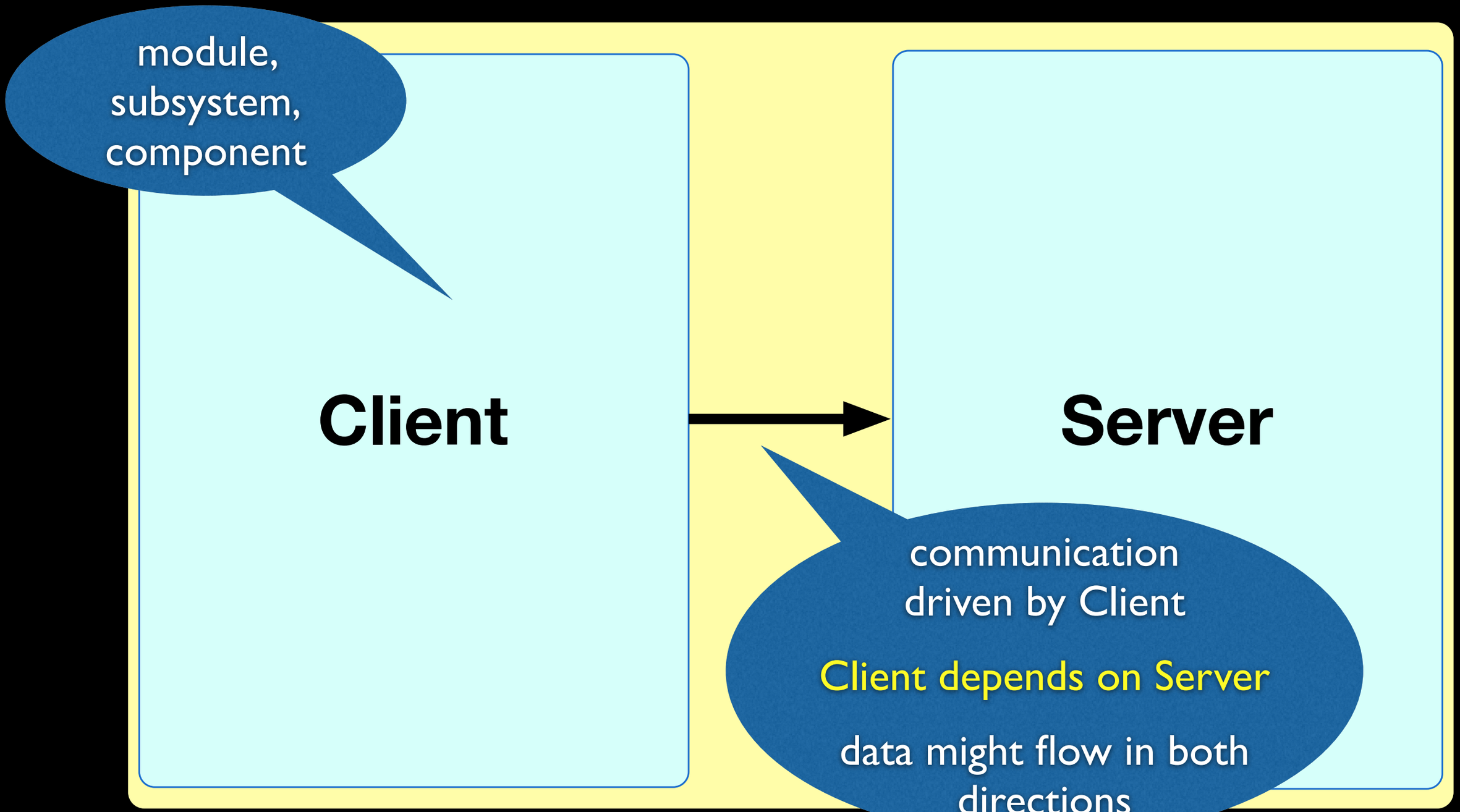
System as a single, atomic entity

System

Decomposition and refinement (adding detail)



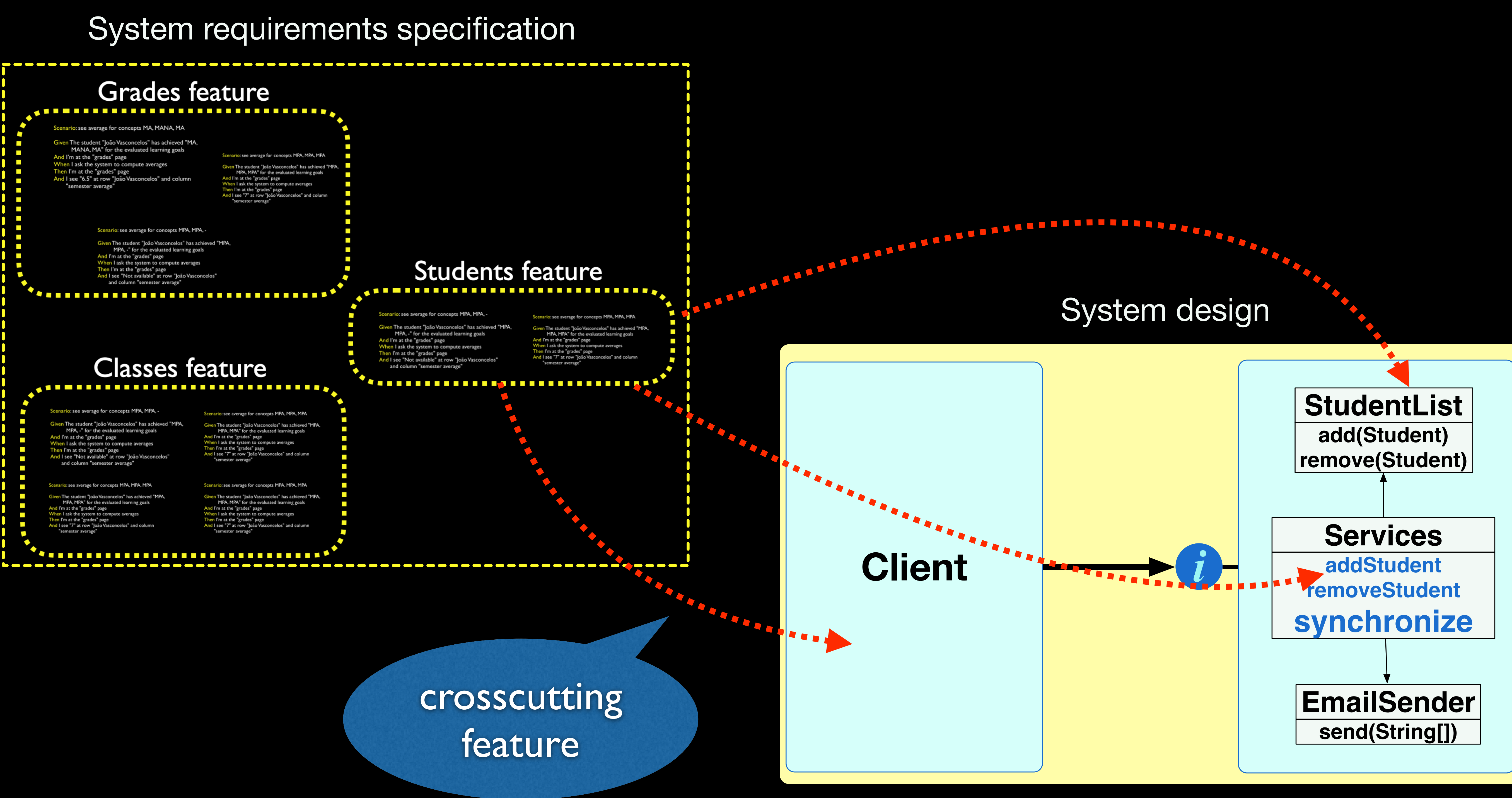
Modules and their communication



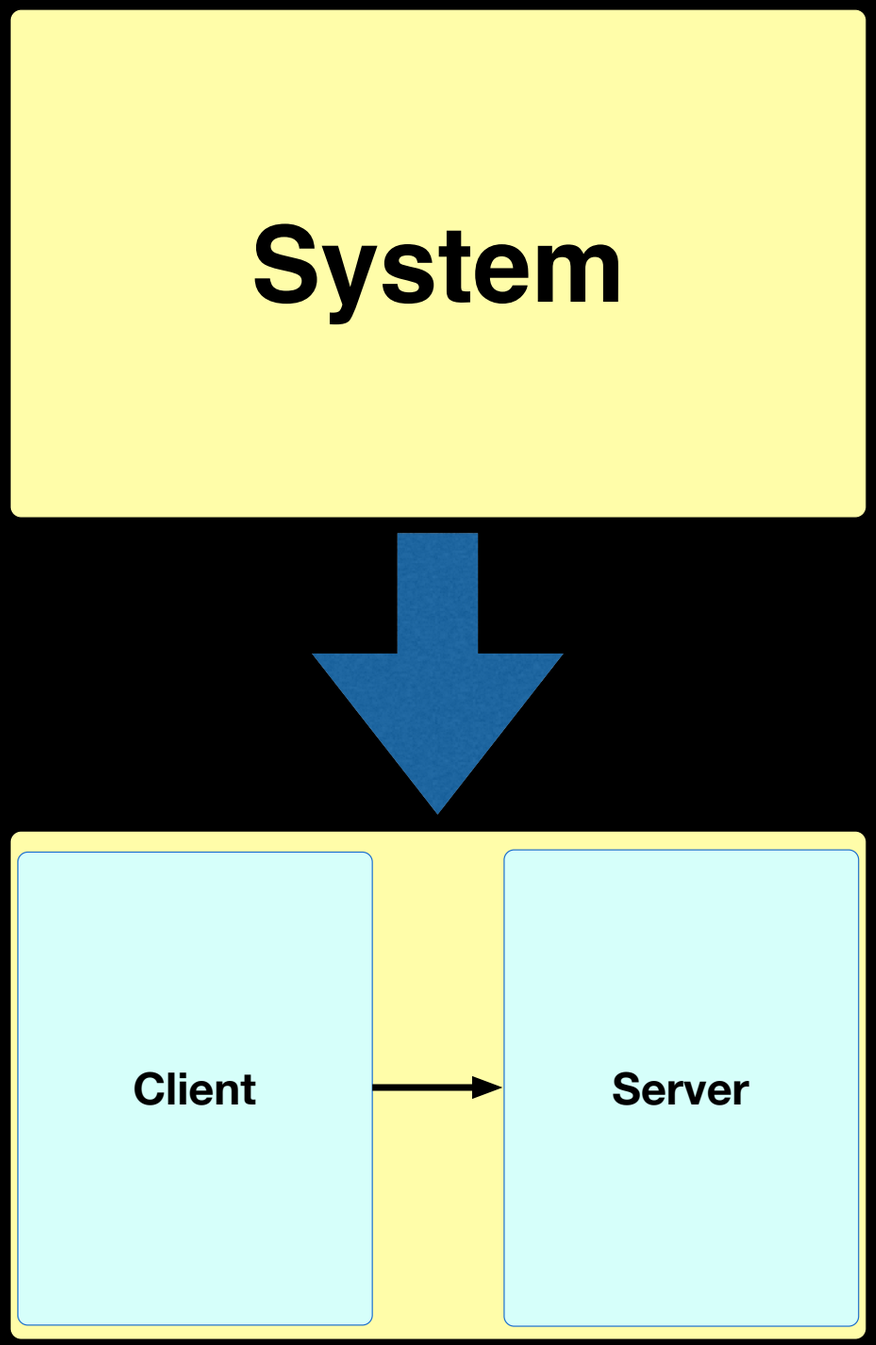
Directed by the existing requirements

aimed at satisfying such requirements

Feature structure and module structure often do not match



Top-down, decomposition



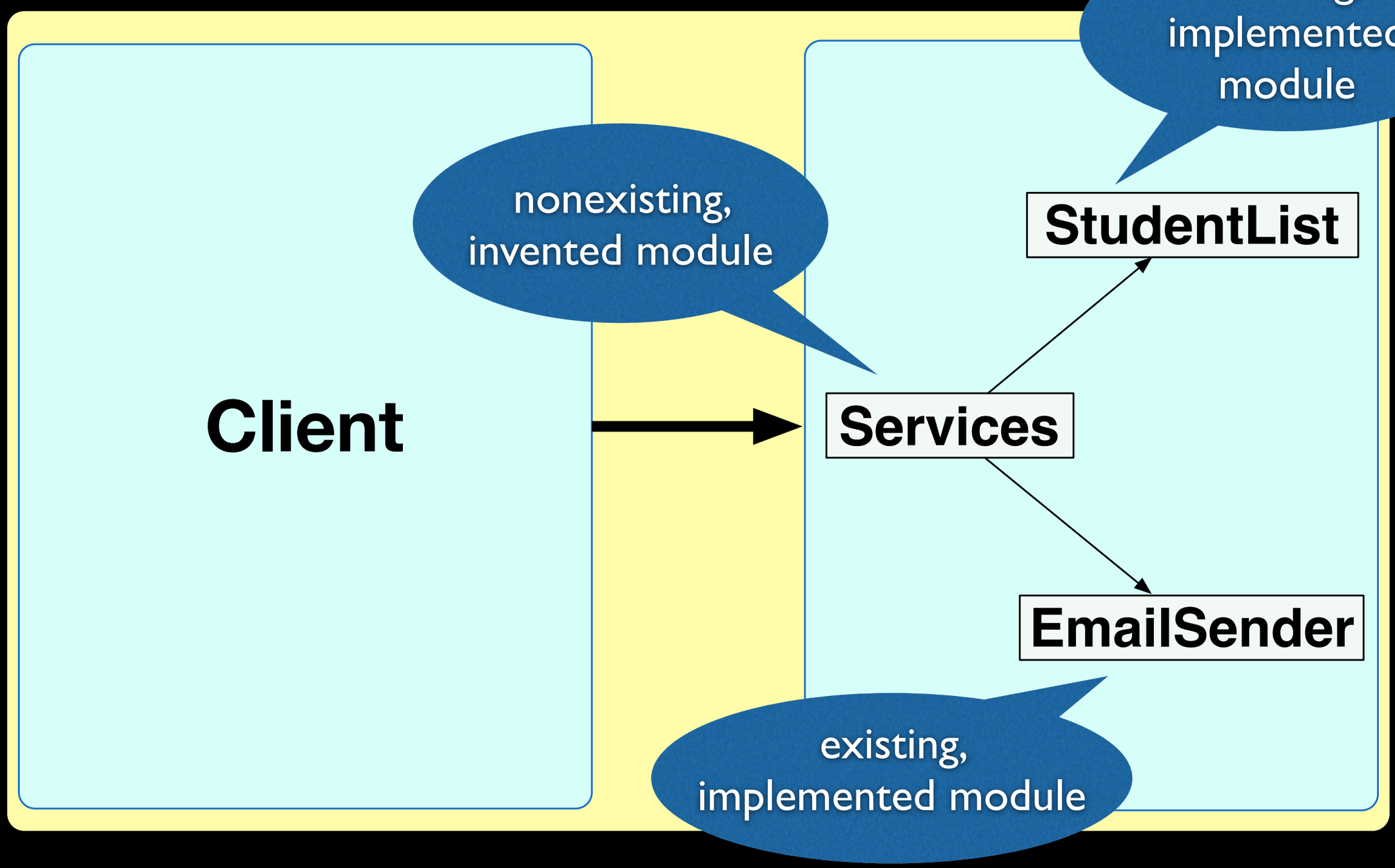
requirements information guides the process

Requirements

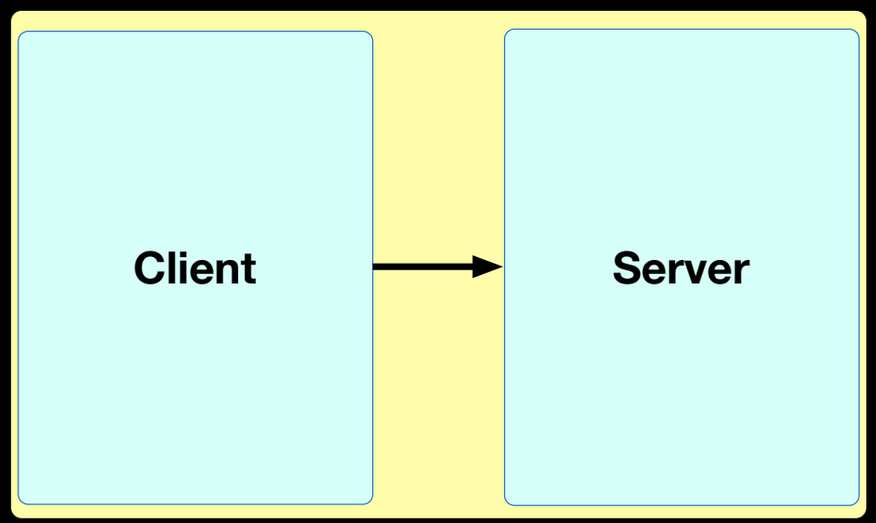
Design

Implementation

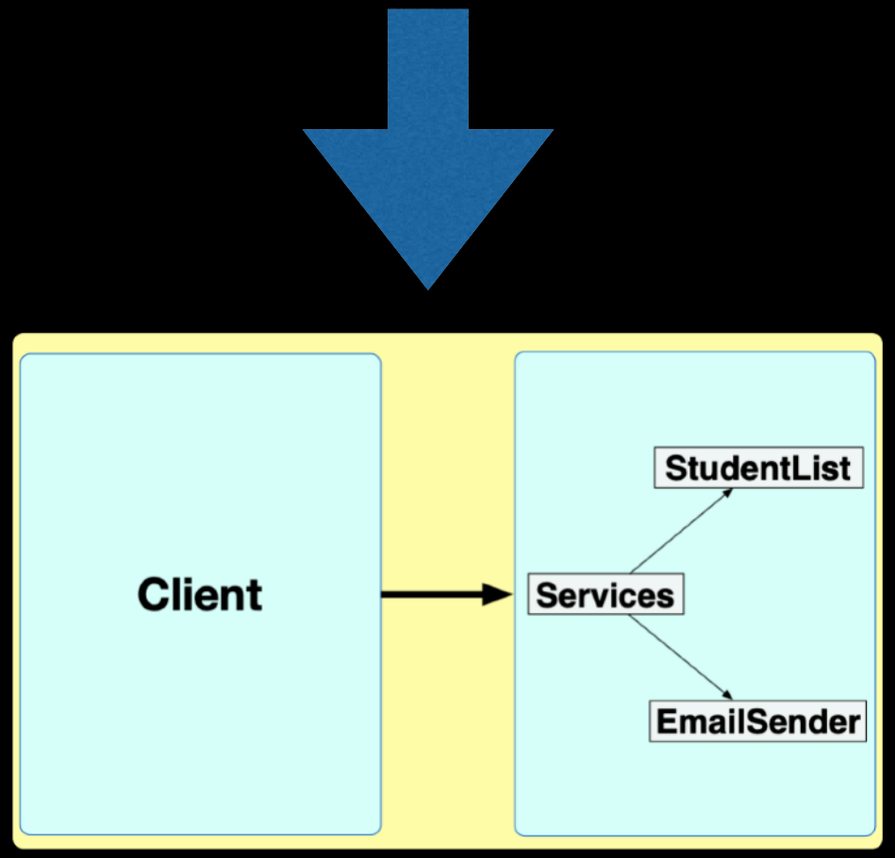
Composition and refinement



Bottom-up, composition



requirements
and
implementation
information
guides the
process



Requirements

Design

Implementation

Design

through…

- Decomposition
- Composition

- Refinement
- Abstraction

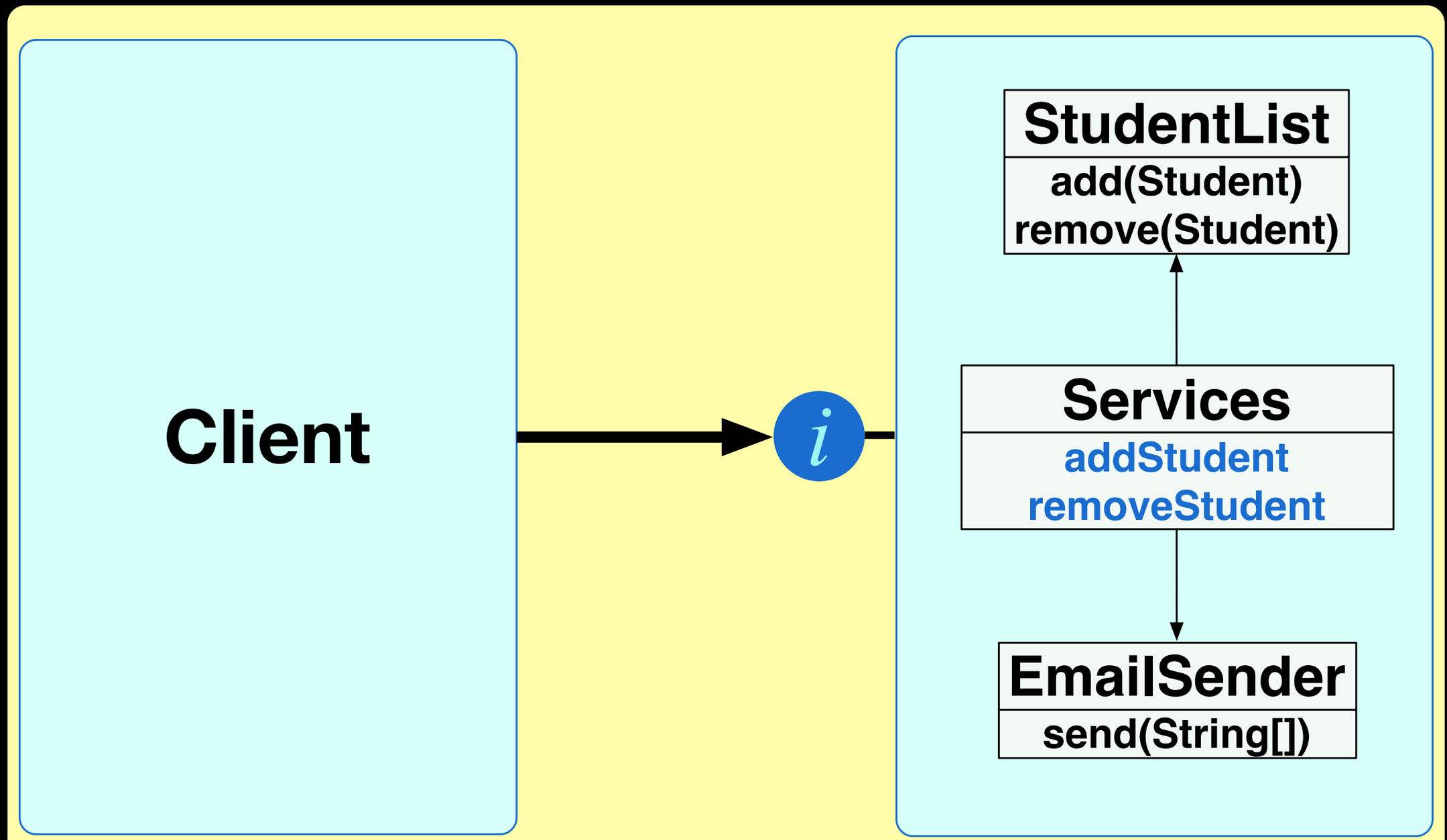
Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

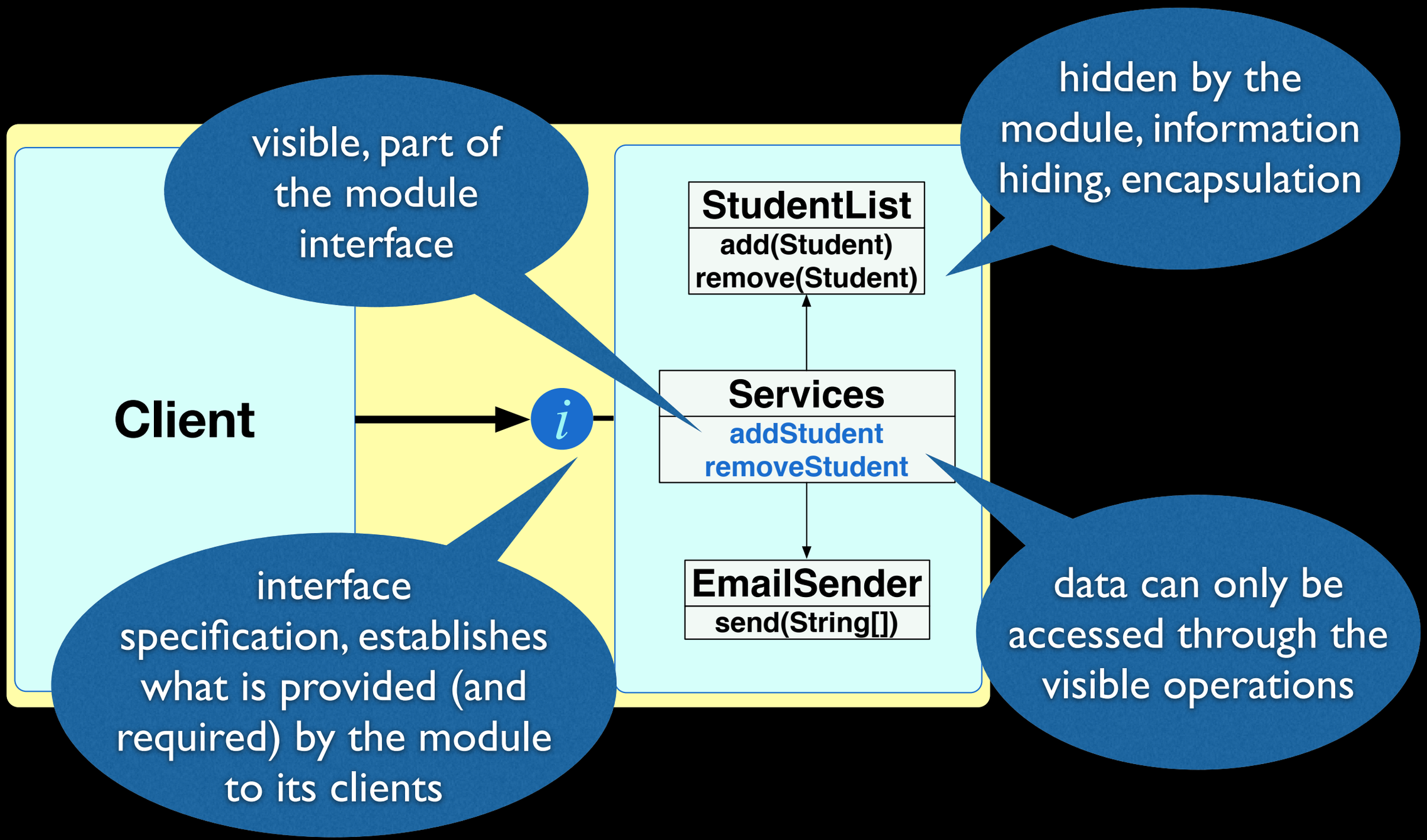
pauloborba.cin.ufpe.br

How to use abstraction during system design?

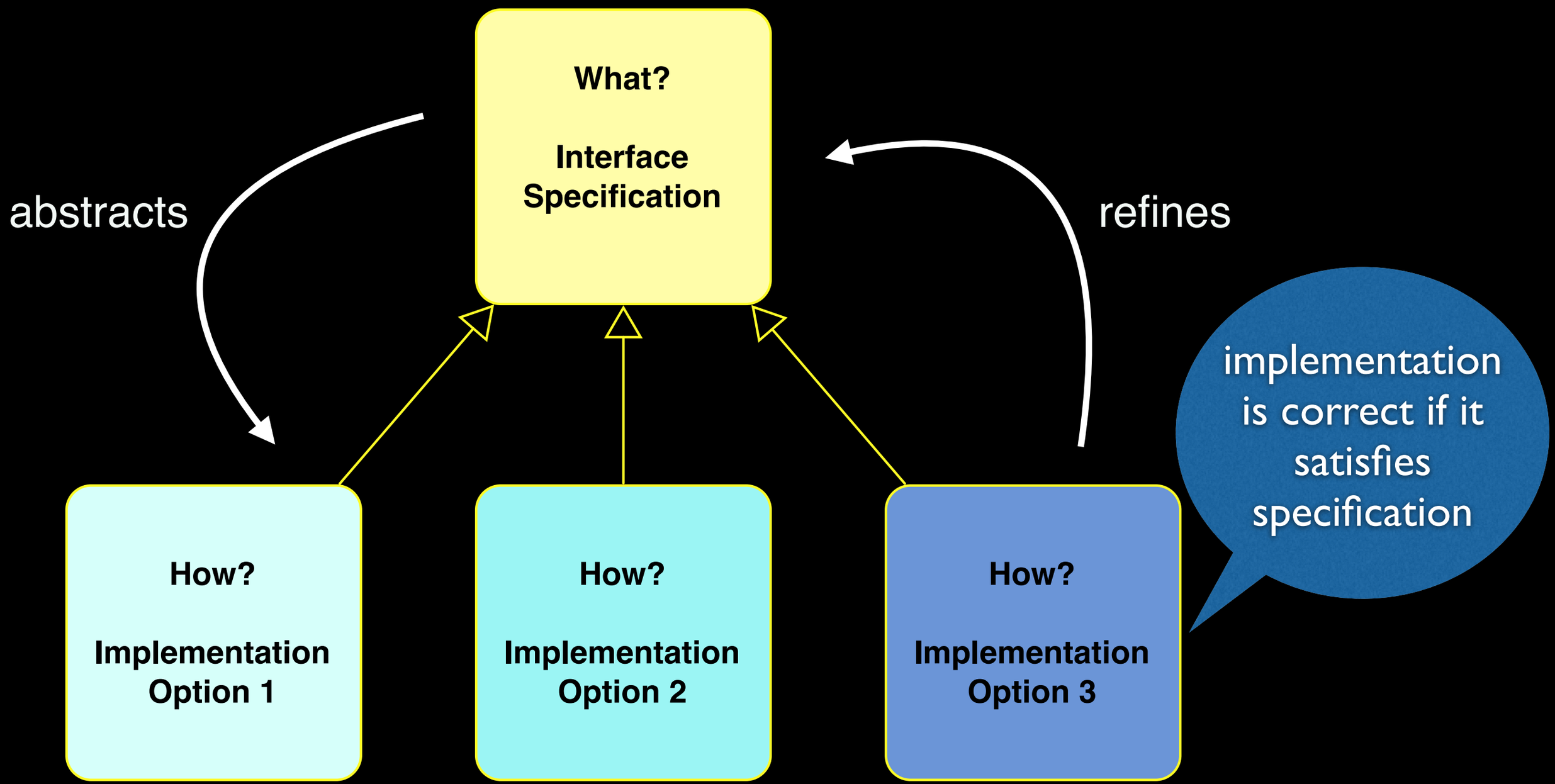
Abstract details that are not necessary for module communication



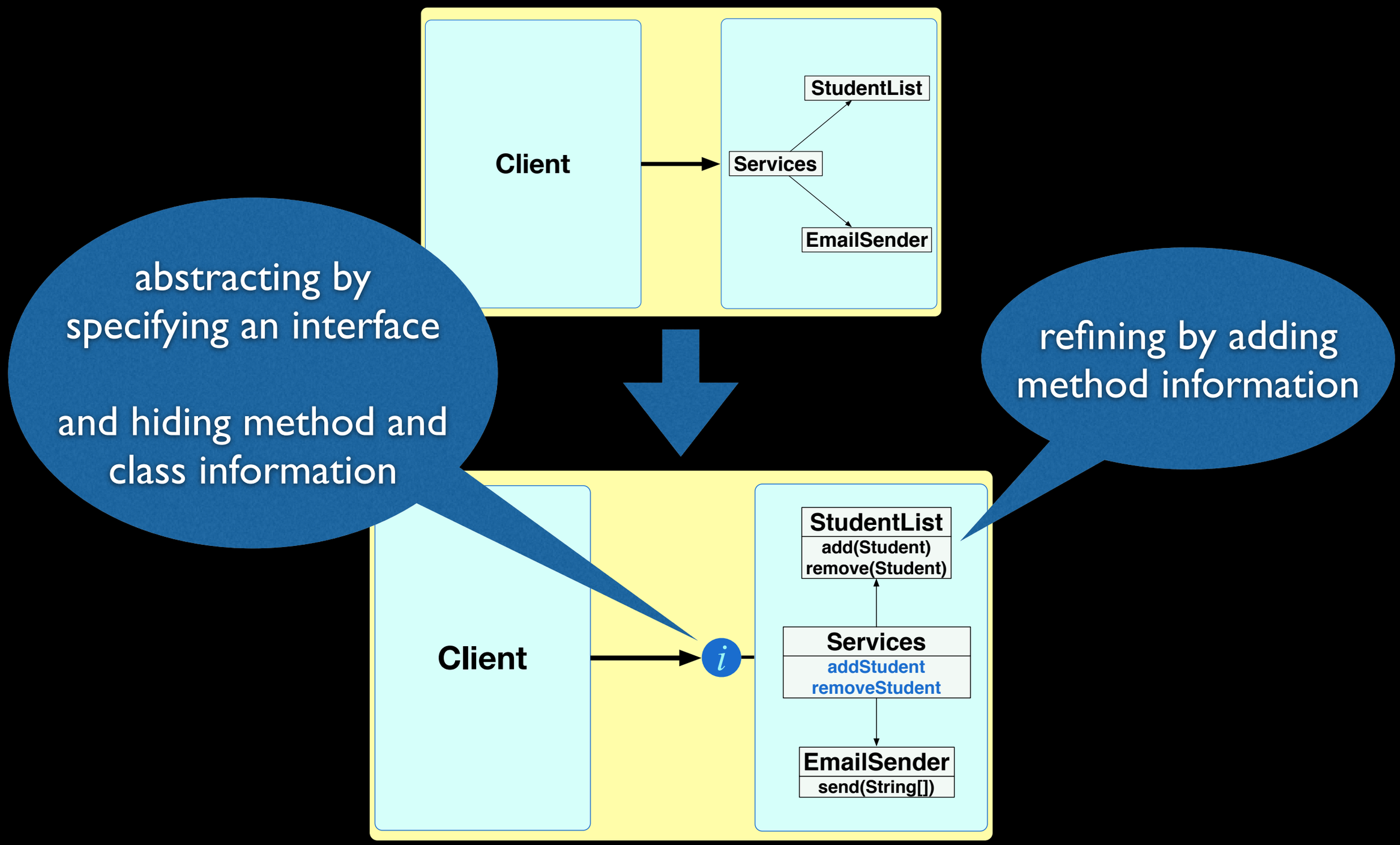
Think of a module as an abstract data type (data + operations) or an abstract machine



Interface reduces dependencies, and enable implementation options



Abstraction and refinement in a single step



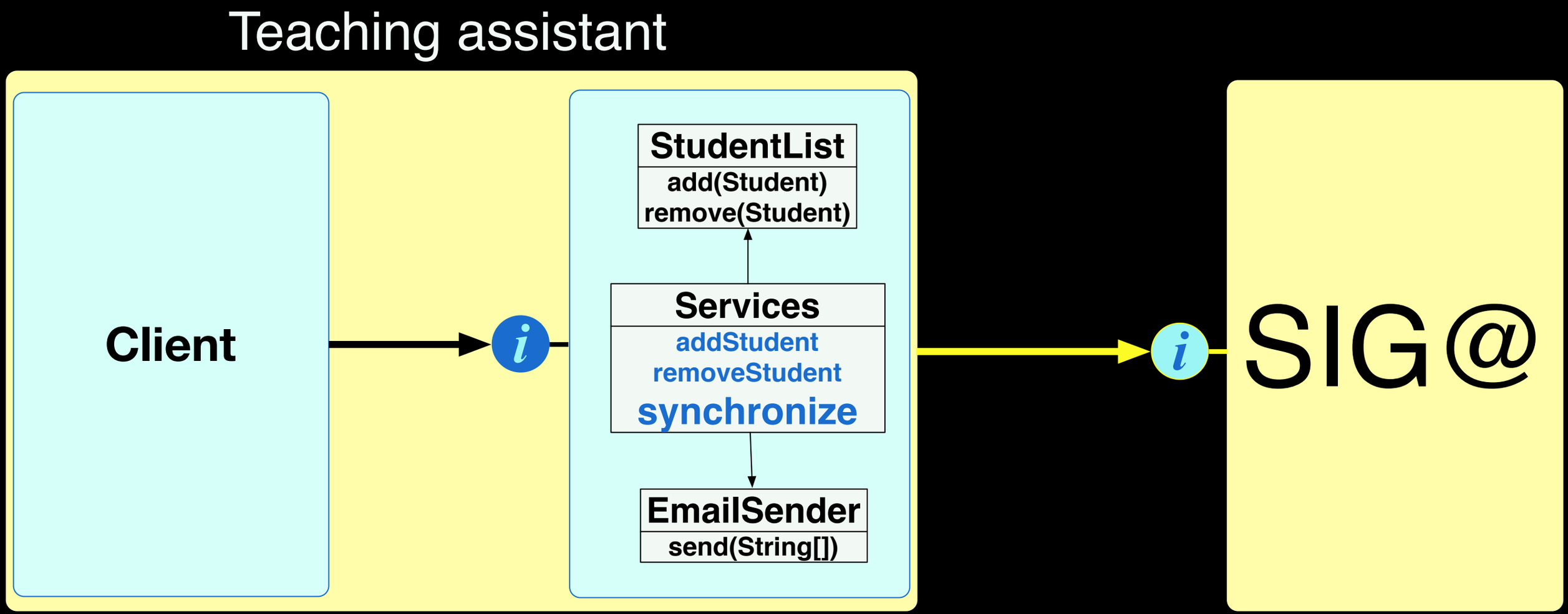
Inventing abstractions to implement modules and systems (collections of modules)

online shopping modules:

browsing (preference data – catalogue data),
ordering (shopping cart and orders data – inventory data),
payment (credit card data – customer data),
delivery (tracking data)

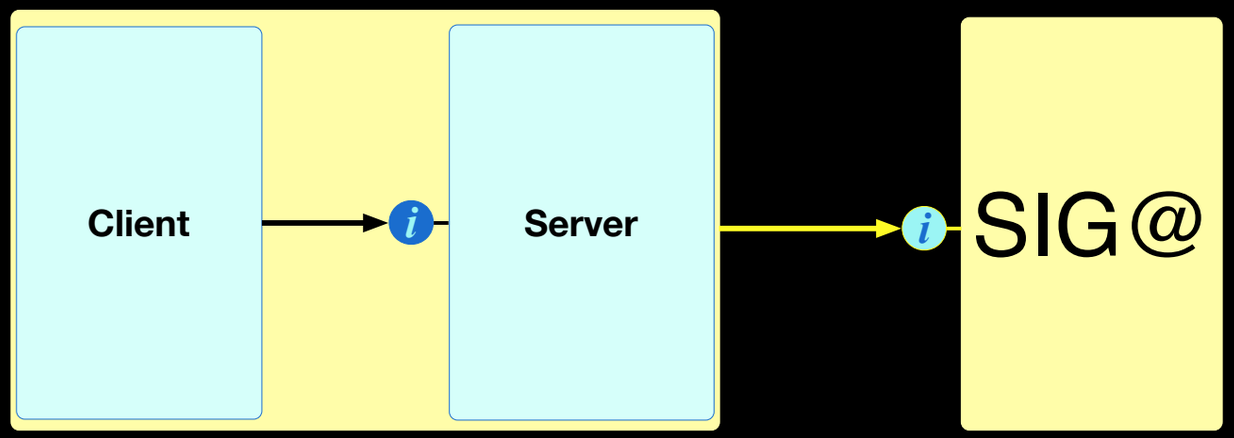
A similar process applies for interacting systems

Uniform approach for internal modules and external systems

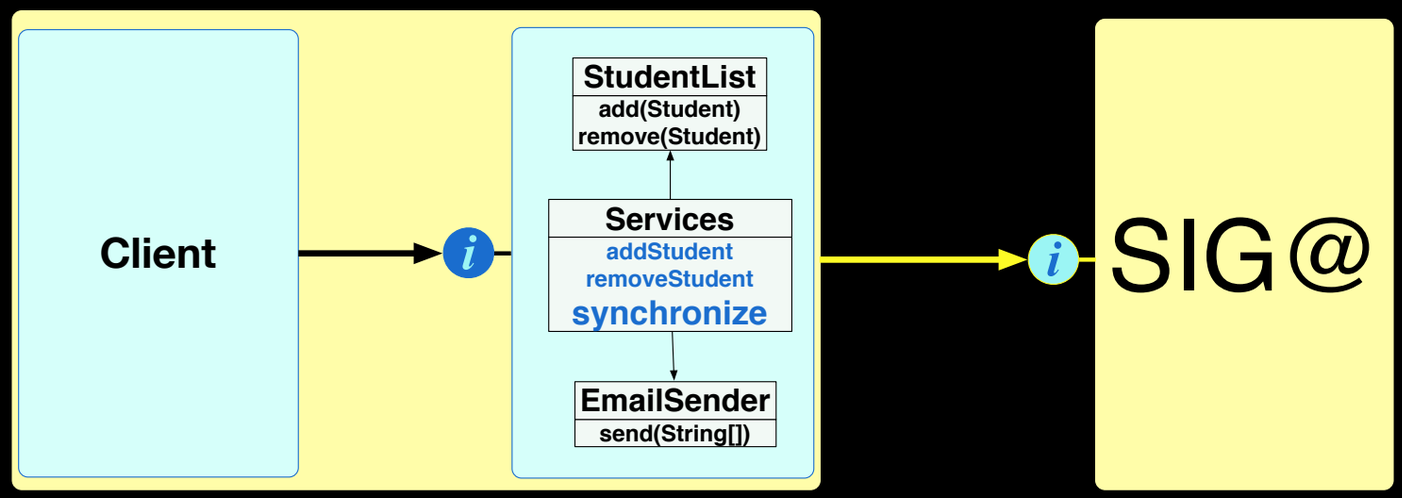

Teaching assistant
Client
StudentList
add(Student)
remove(Student)
Services
addStudent
removeStudent
synchronize
EmailSender
send(String[])
SIG@

Design at different levels

Architecture design



High level design

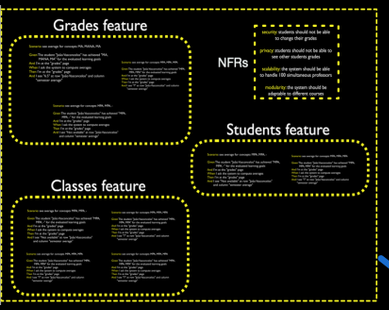


Low level design

Code

The design process is a decision making process, reducing implementation options, with the aim of satisfying the system requirements

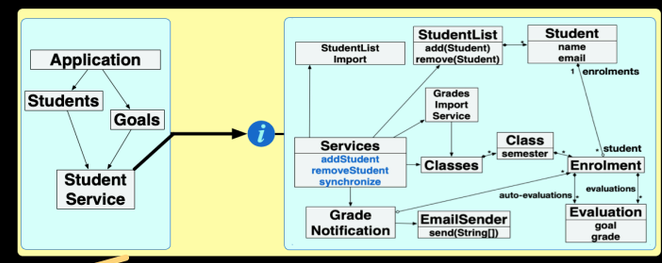
restricts system external behavior



All systems

All systems sastifying requirements

All systems sastifying requirements and design model



restricts system internal structure (and behavior)

All systems

All systems sastifying requirements

All systems sastifying requirements and design model

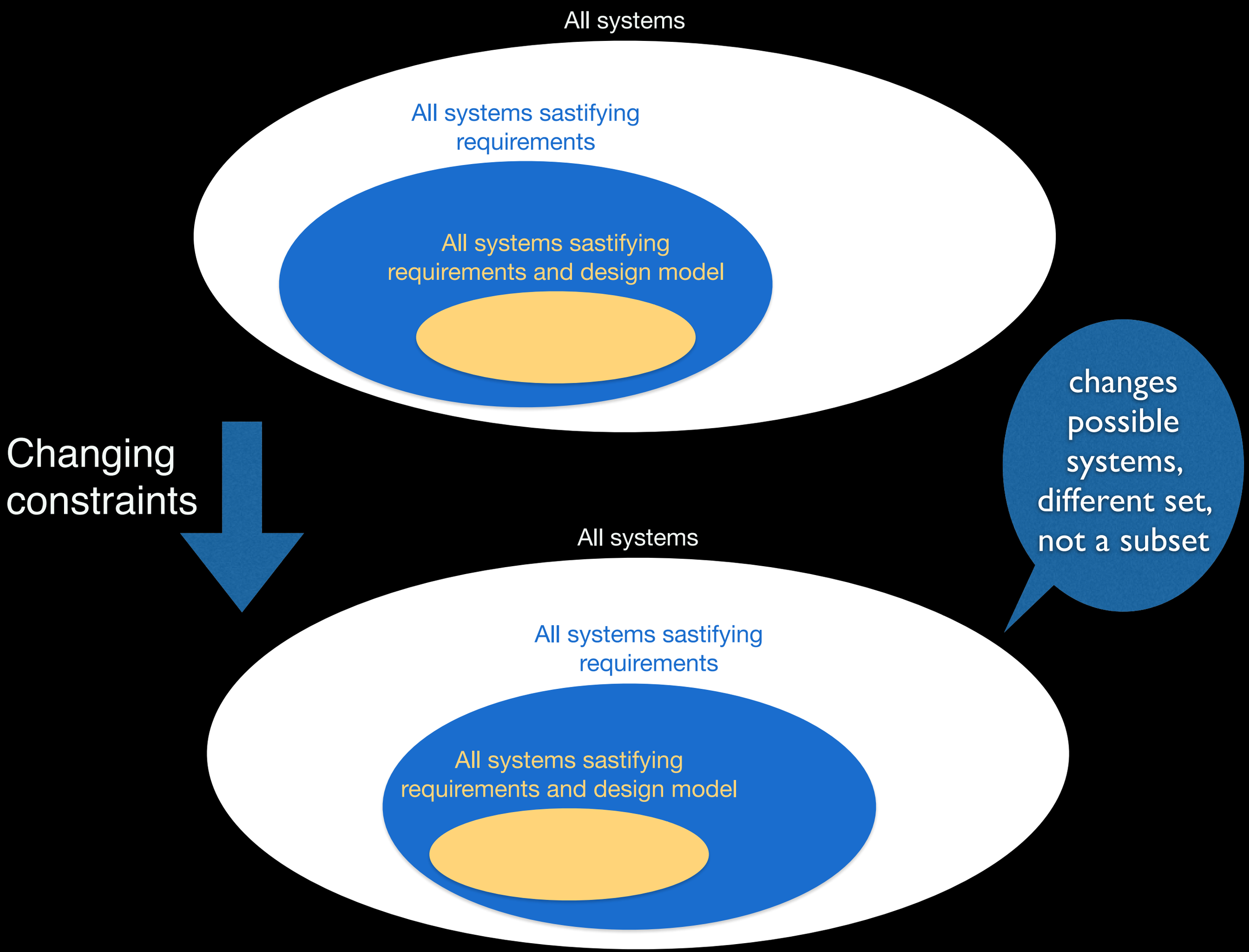
Adding constraints

reduces possible systems, subset

All systems

All systems sastifying requirements

All systems sastifying requirements and design model


All systems
All systems sastifying requirements
All systems sastifying requirements and design model
Changing constraints
changes possible systems, different set, not a subset
All systems
All systems sastifying requirements
All systems sastifying requirements and design model

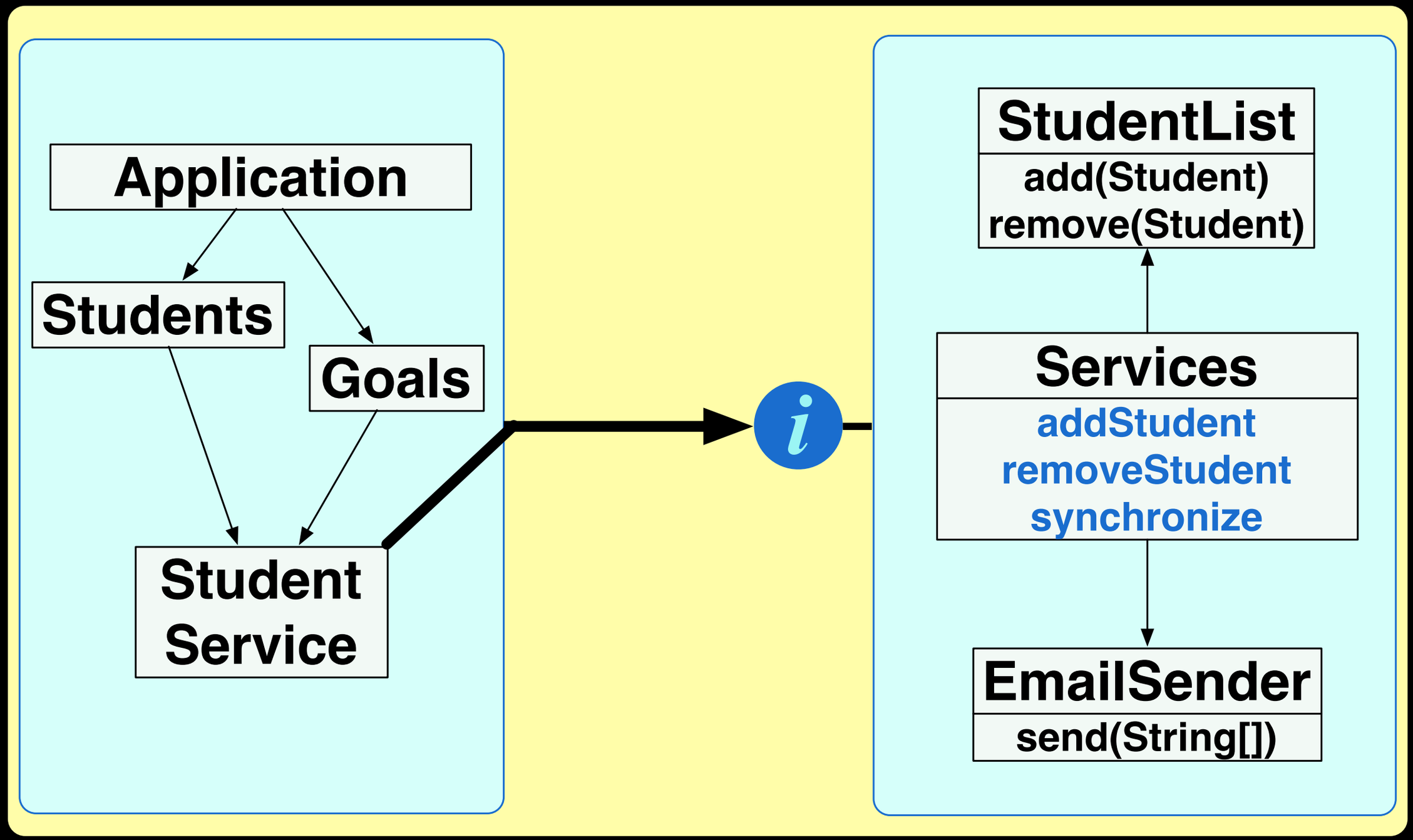
Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

How to detail the design with domain driven design (DDD)?

Detailing the design of the client



Design rationale

One main module for the main menu

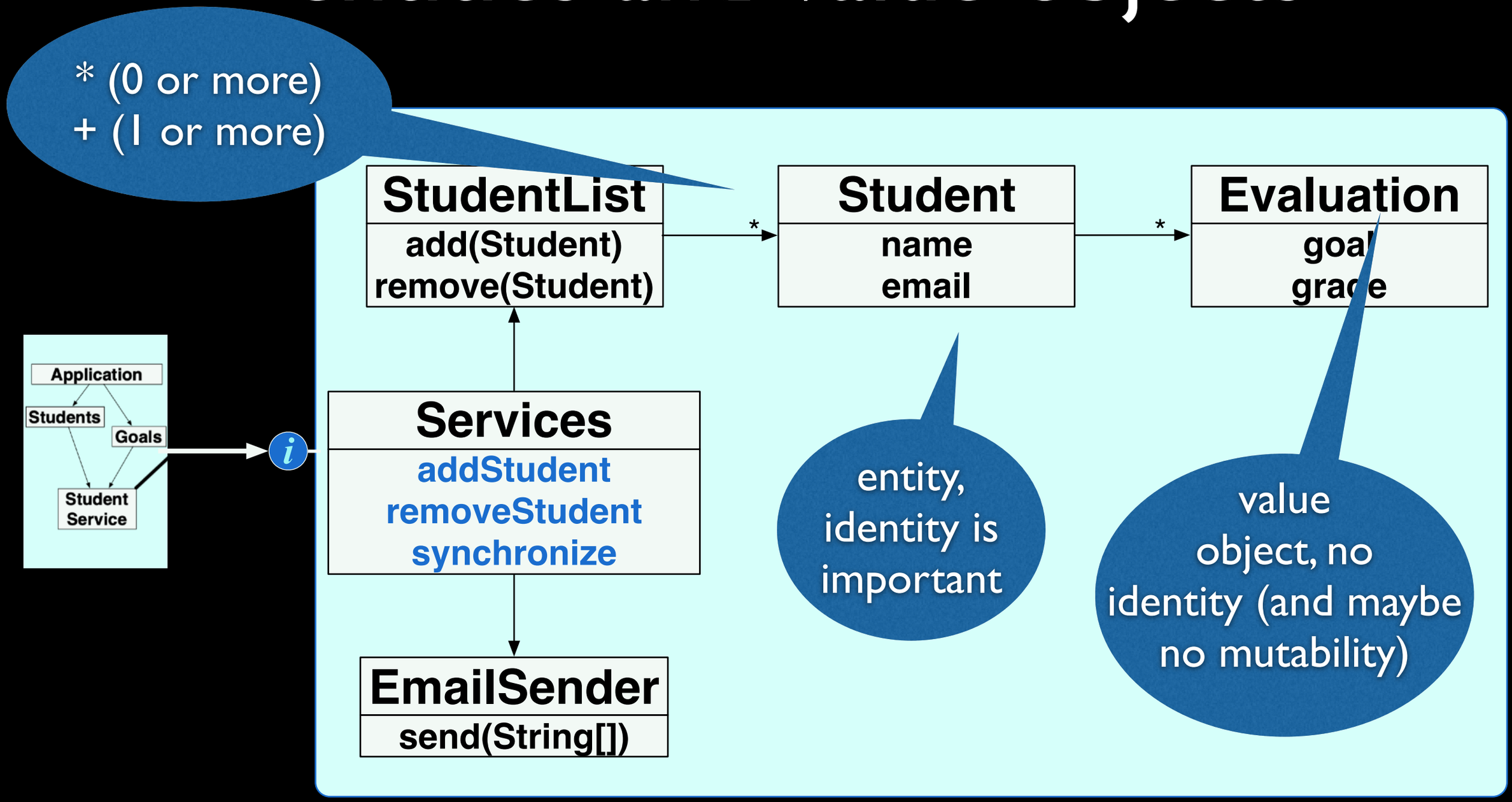
One module per menu option (tab view)

Could be further decomposed, for improving extensibility (readability) or enabling reuse

One module with common services

Could have others, to separate concerns and improve internal quality as well

Detailing the design of the server by identifying related entities and value objects



Design rationale

One module for Student because it's an important concept

Could be avoided if directly accessing a database, and returning to the client only partial or unstructured information

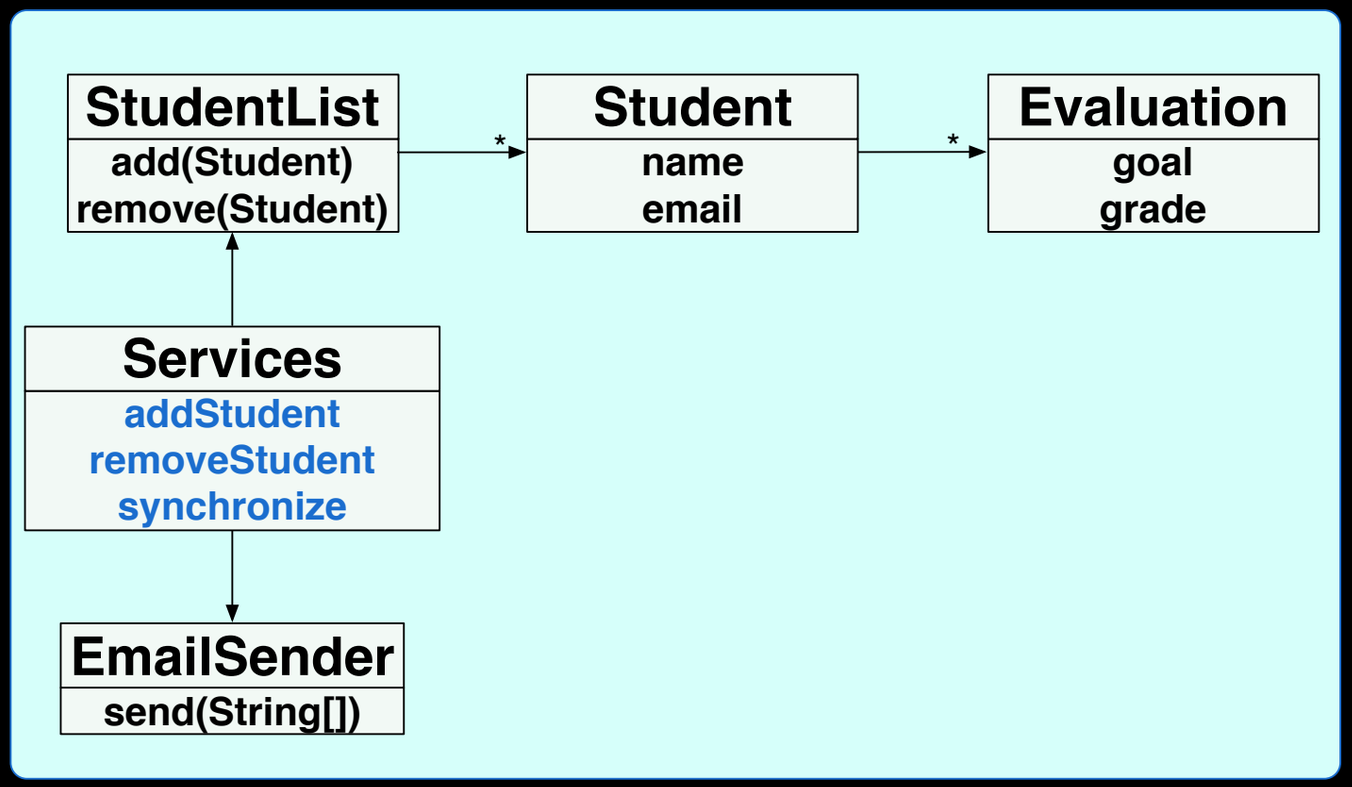
One module for Evaluation for separation of concerns

Could be avoided by being indirectly represented inside Student, possibly compromising readability

compromises performance? battery usage?

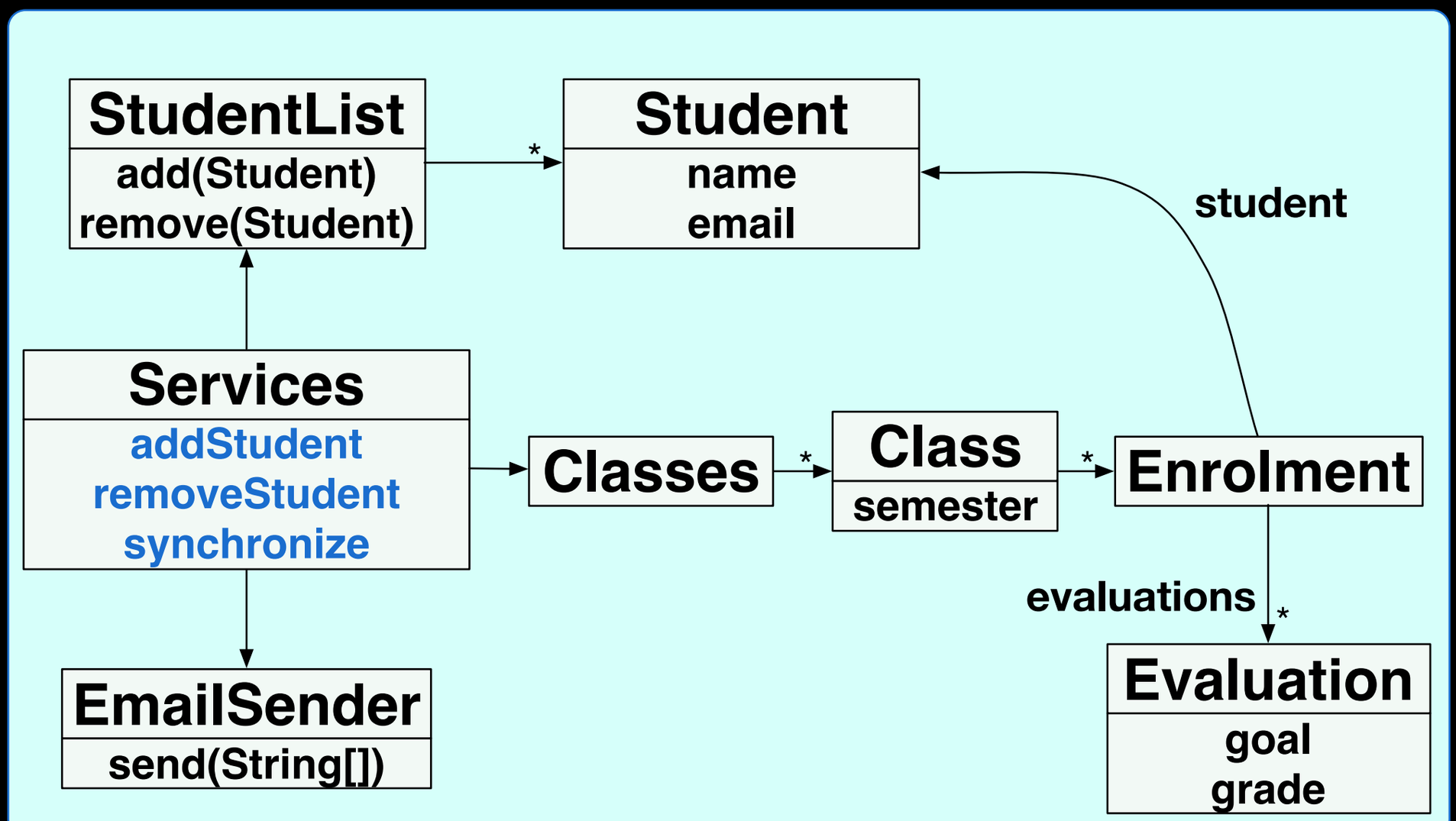
Driven by the specified requirements

If the system is to be used every semester, the current design does not keep grade information for a student who failed in a previous semester



First design

Second design

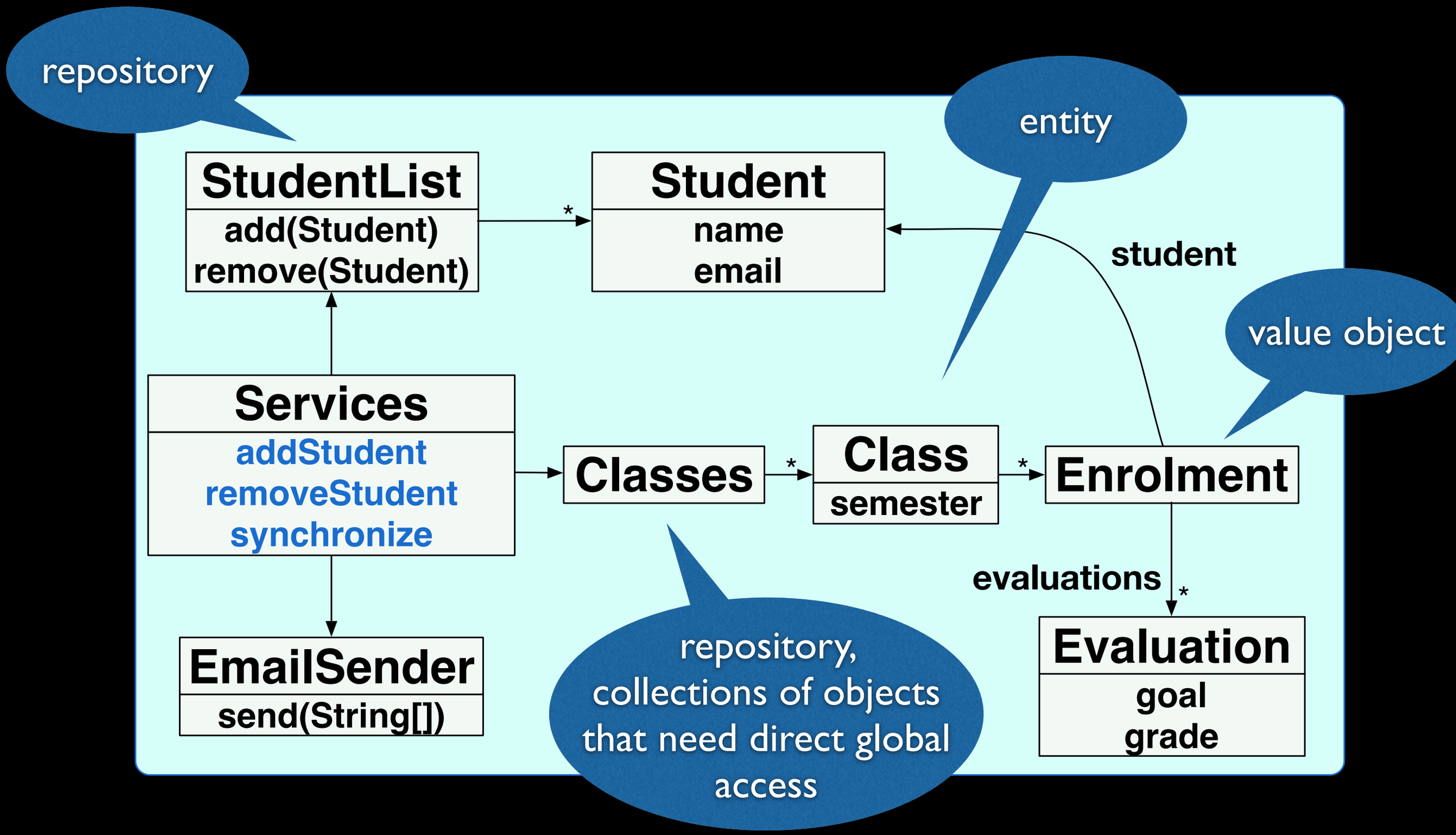


The best design?

Does it satisfy the requirements?

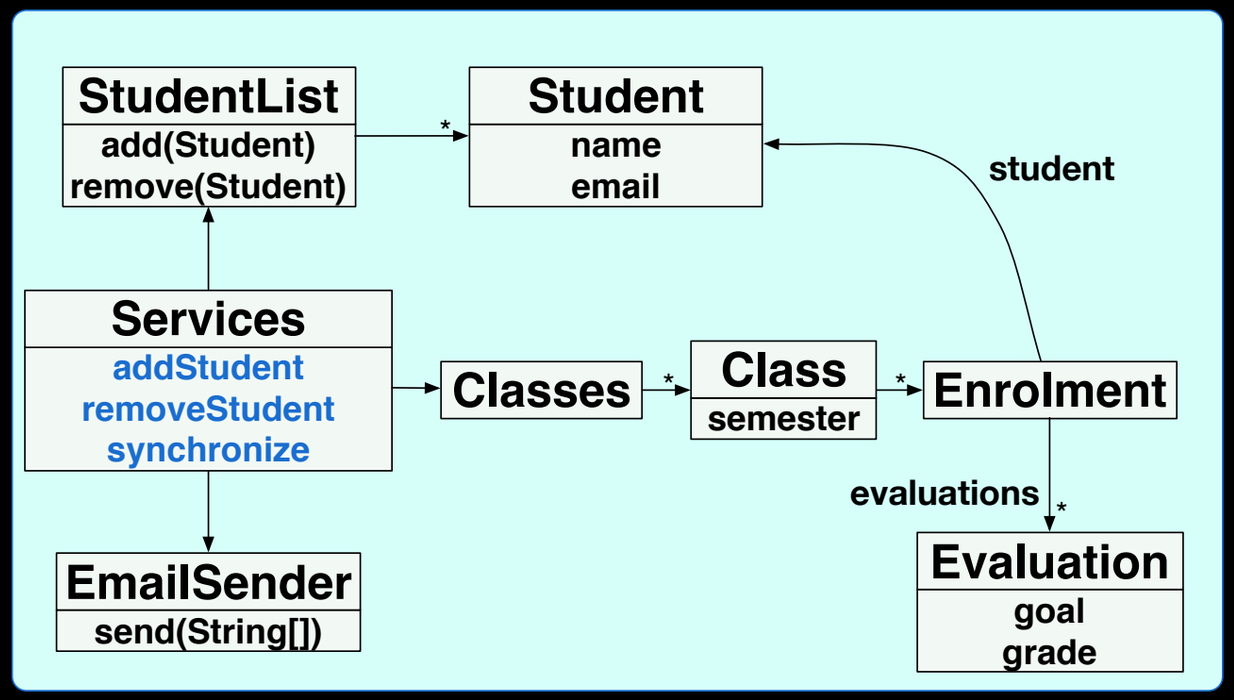
How strongly does it satisfy the requirements?

Identifying repositories



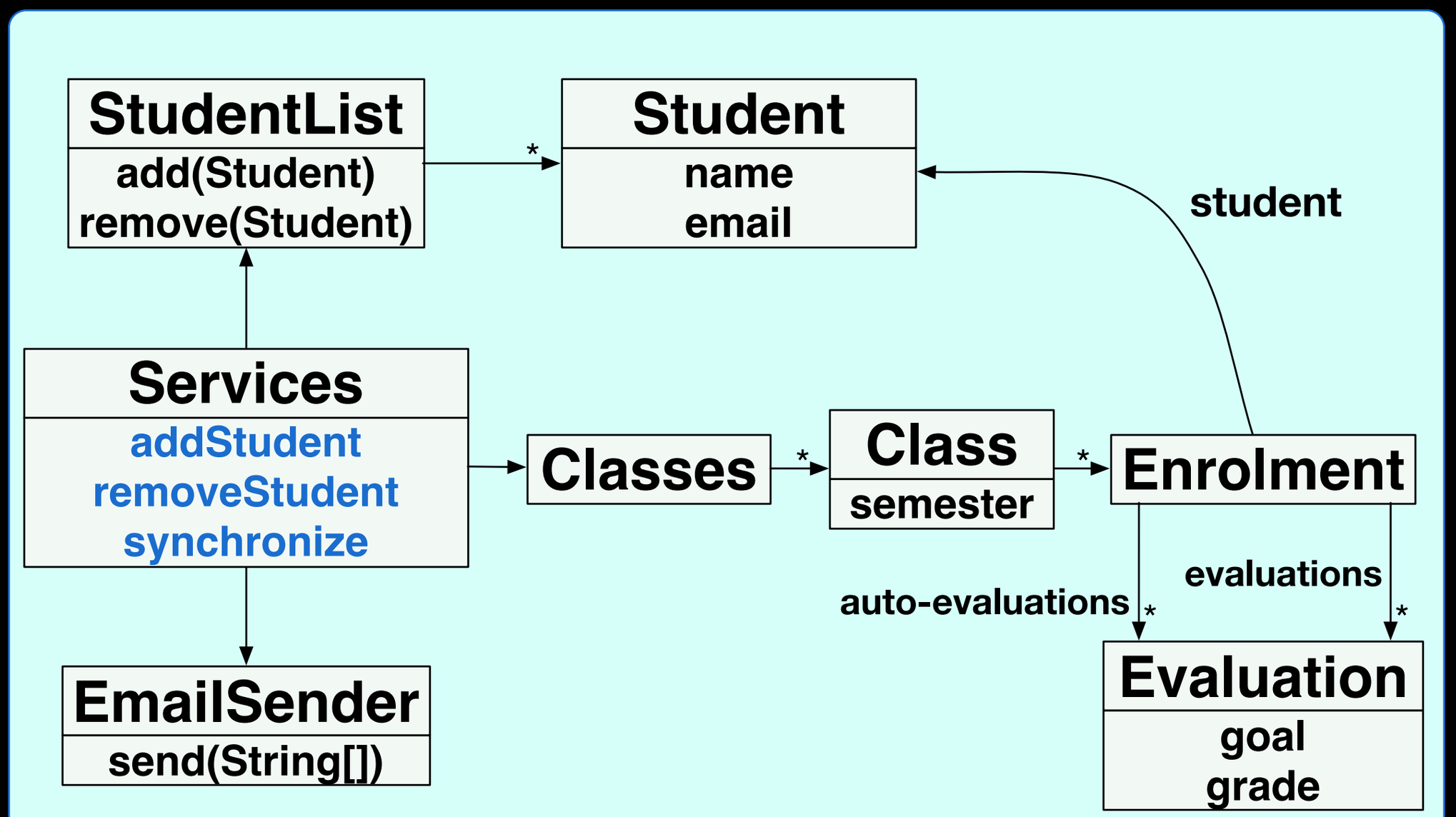
All driven by the (specified) system requirements

Are the stakeholders interested in an auto-evaluation feature?

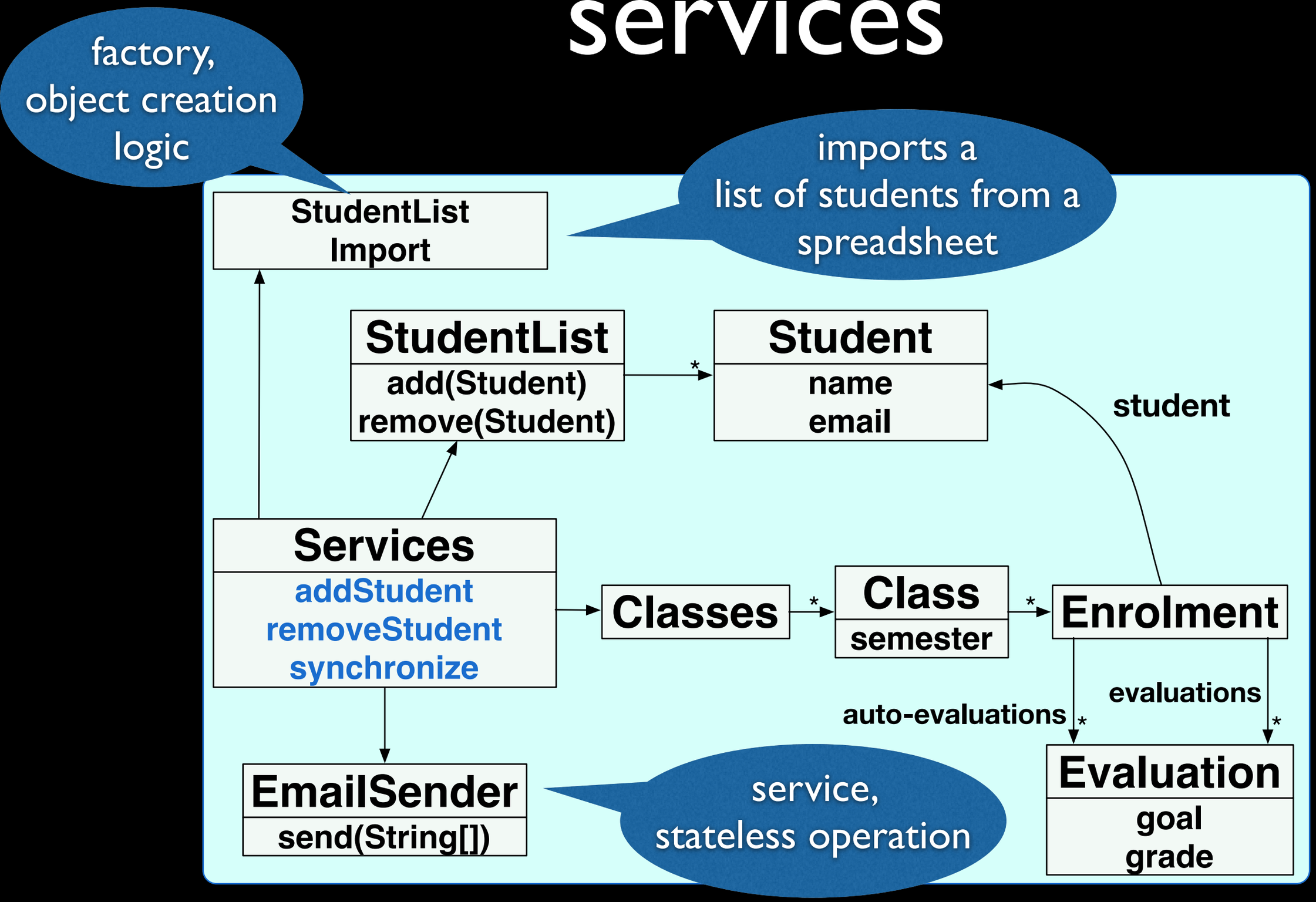

StudentList
add(Student)
remove(Student)
*
Student
name
email
student
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
Enrolment
evaluations
*
Evaluation
goal
grade
EmailSender
send(String[])

No

Yes

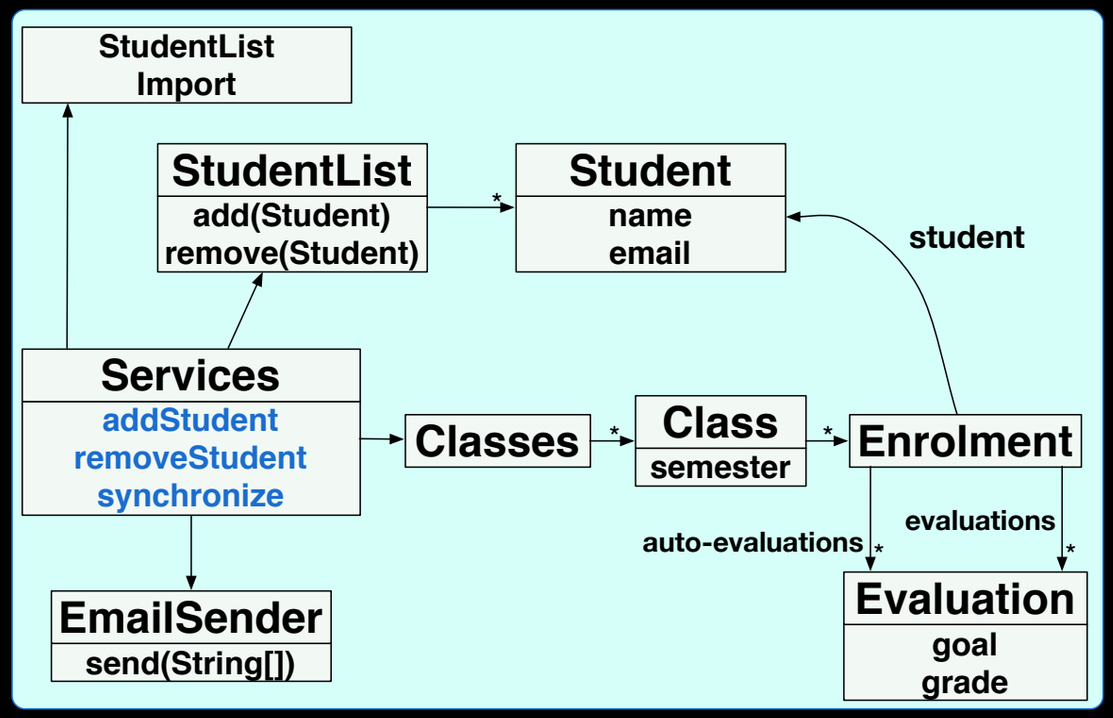

StudentList
add(Student)
remove(Student)
*
Student
name
email
student
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
Enrolment
auto-evaluations
*
evaluations
*
Evaluation
goal
grade
EmailSender
send(String[])

Identifying factories and services



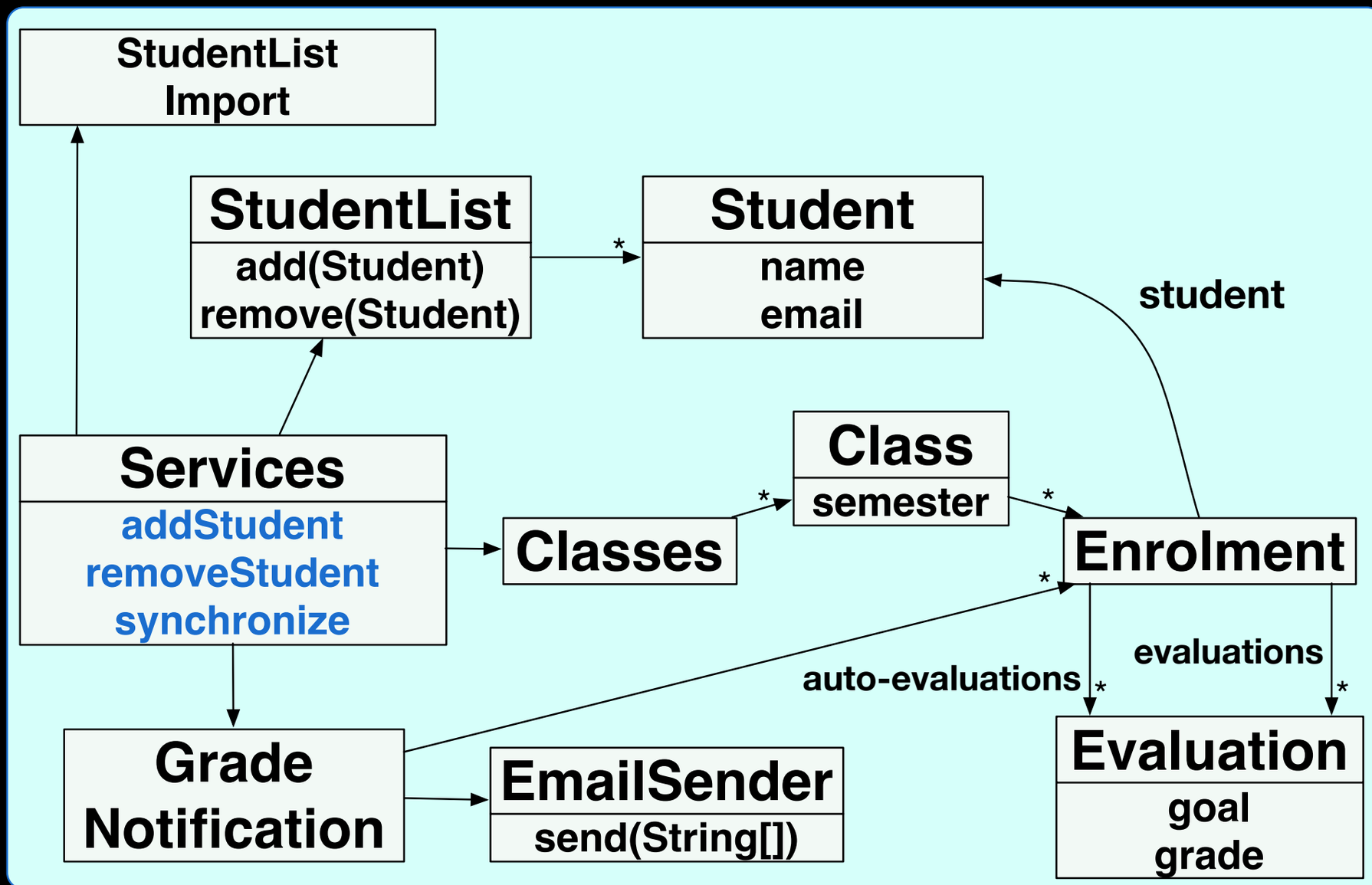
All driven by the (specified) system requirements

Should notification be sent only once a day?

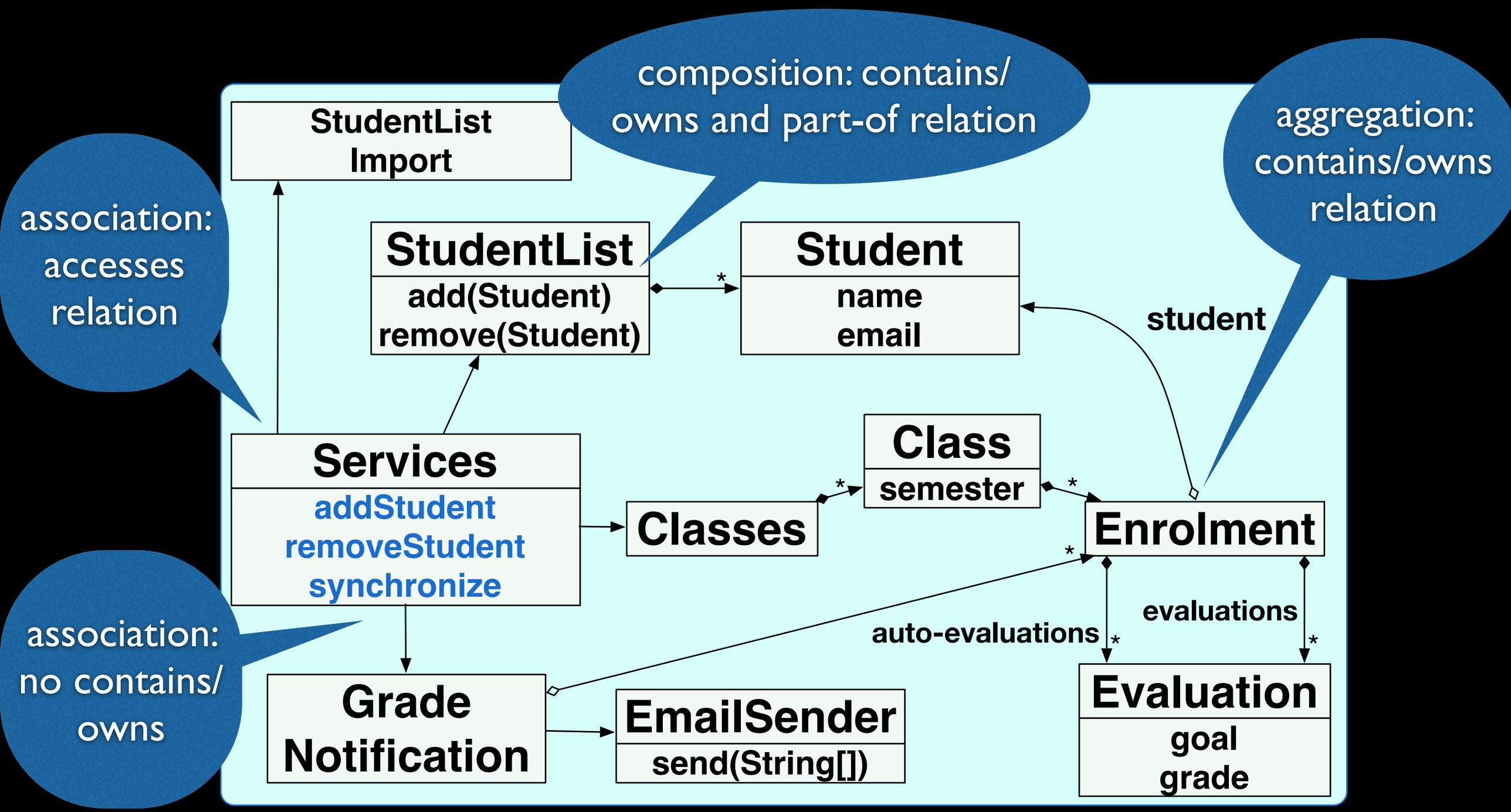

StudentList
Import
StudentList
add(Student)
remove(Student)
*
Student
name
email
student
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
Enrolment
auto-evaluations
*
evaluations
*
EmailSender
send(String[])
Evaluation
goal
grade

No

Yes

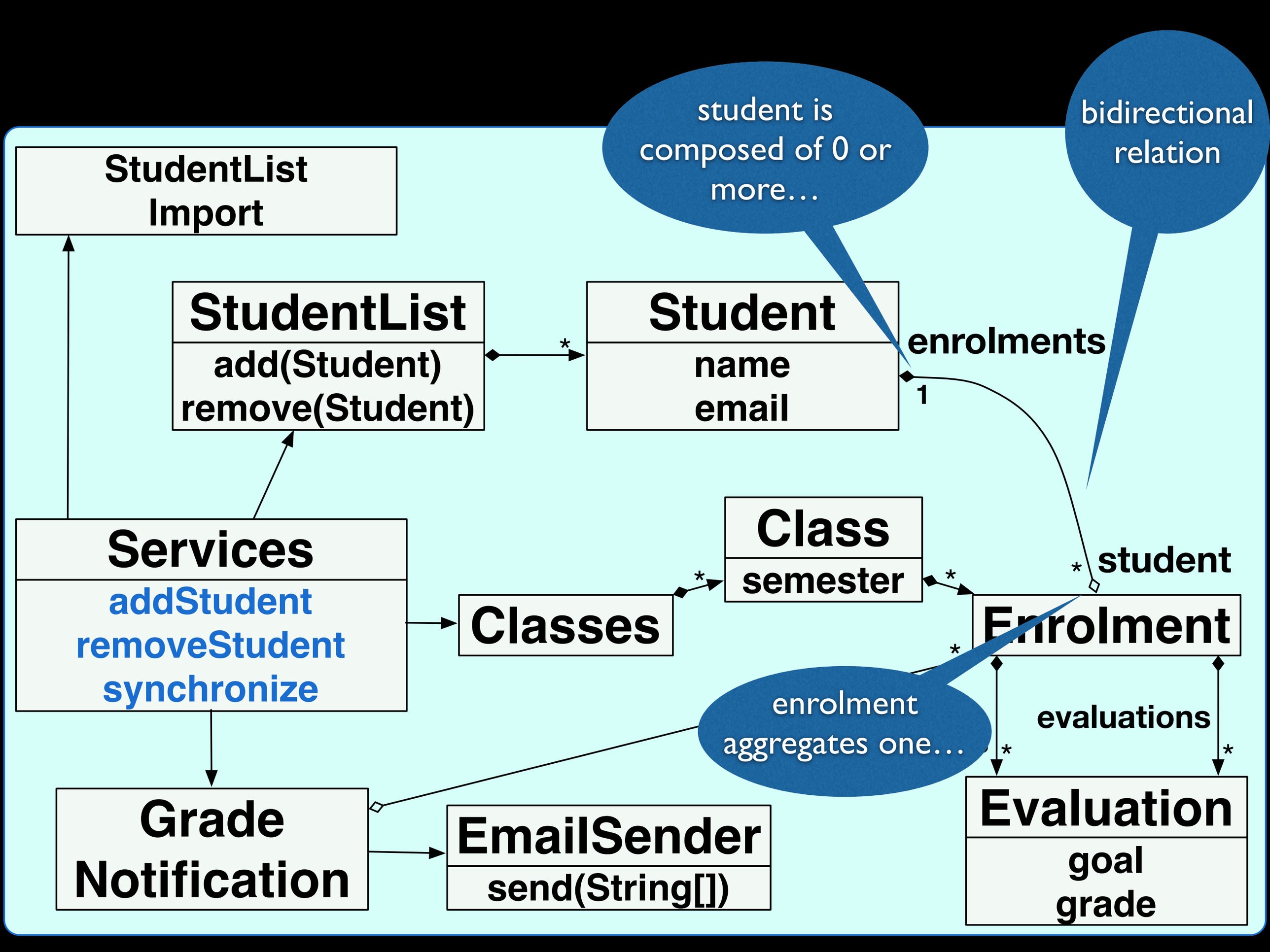

StudentList
Import
StudentList
add(Student)
remove(Student)
*
Student
name
email
student
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
Enrolment
auto-evaluations
*
evaluations
*
Grade
Notification
EmailSender
send(String[])
Evaluation
goal
grade

Composition, aggregation, and association information



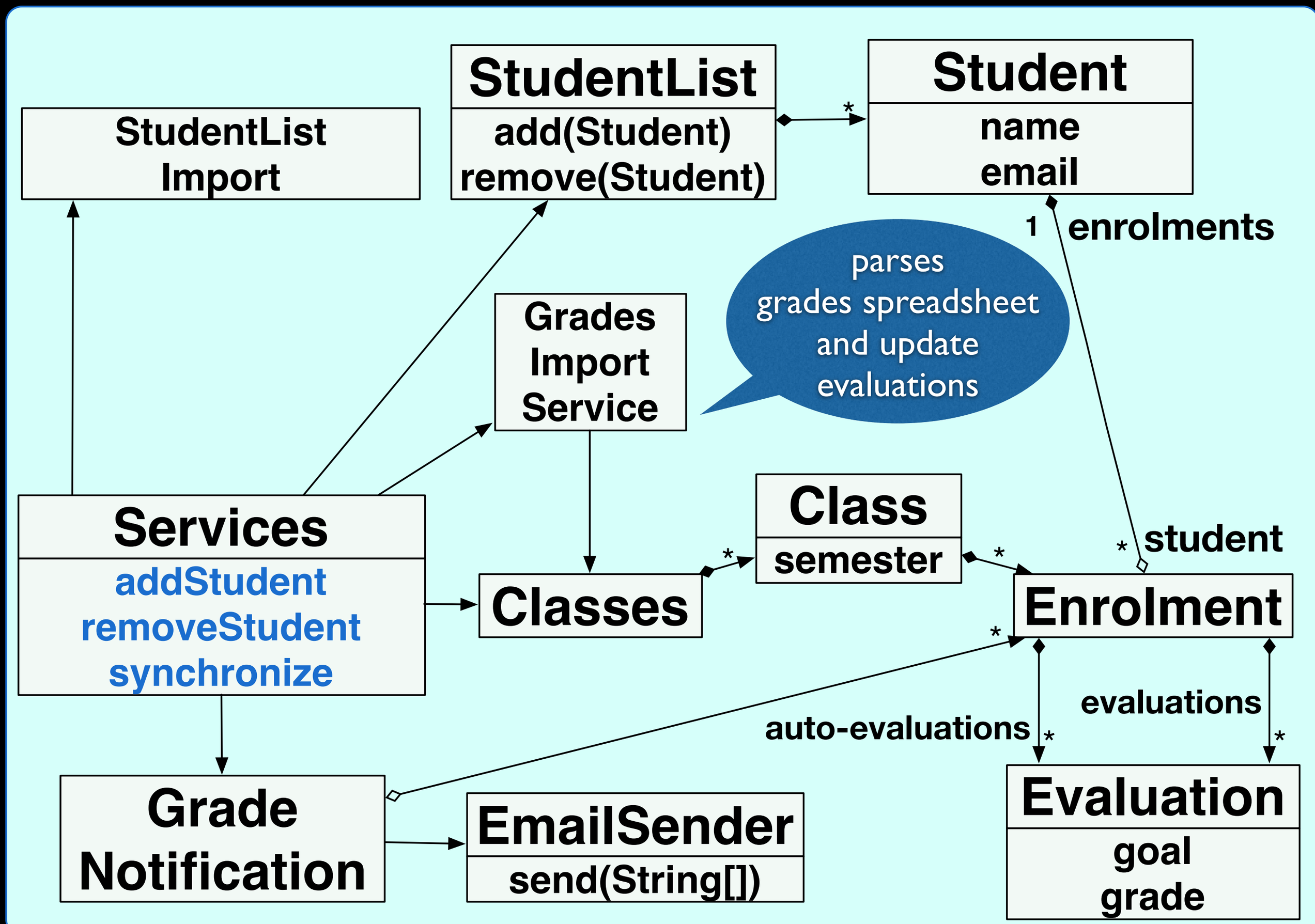
All driven by the (specified) system requirements

If there is a need to show all enrolments from a given student

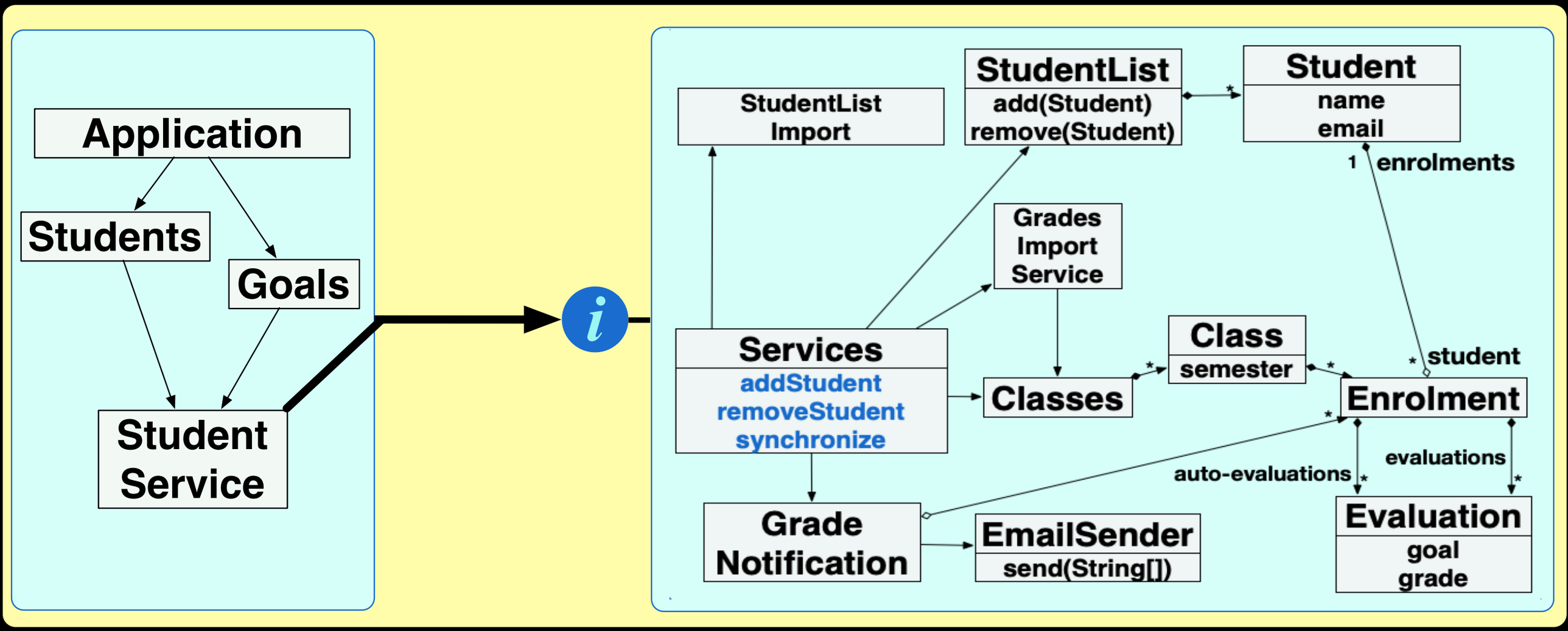

StudentList
Import
StudentList
add(Student)
remove(Student)
*
Student
name
email
enrolments
1
student is composed of 0 or more…
bidirectional relation
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
*
student
Enrolment
*
enrolment aggregates one…
evaluations
*
*
Grade
Notification
EmailSender
send(String[])
Evaluation
goal
grade

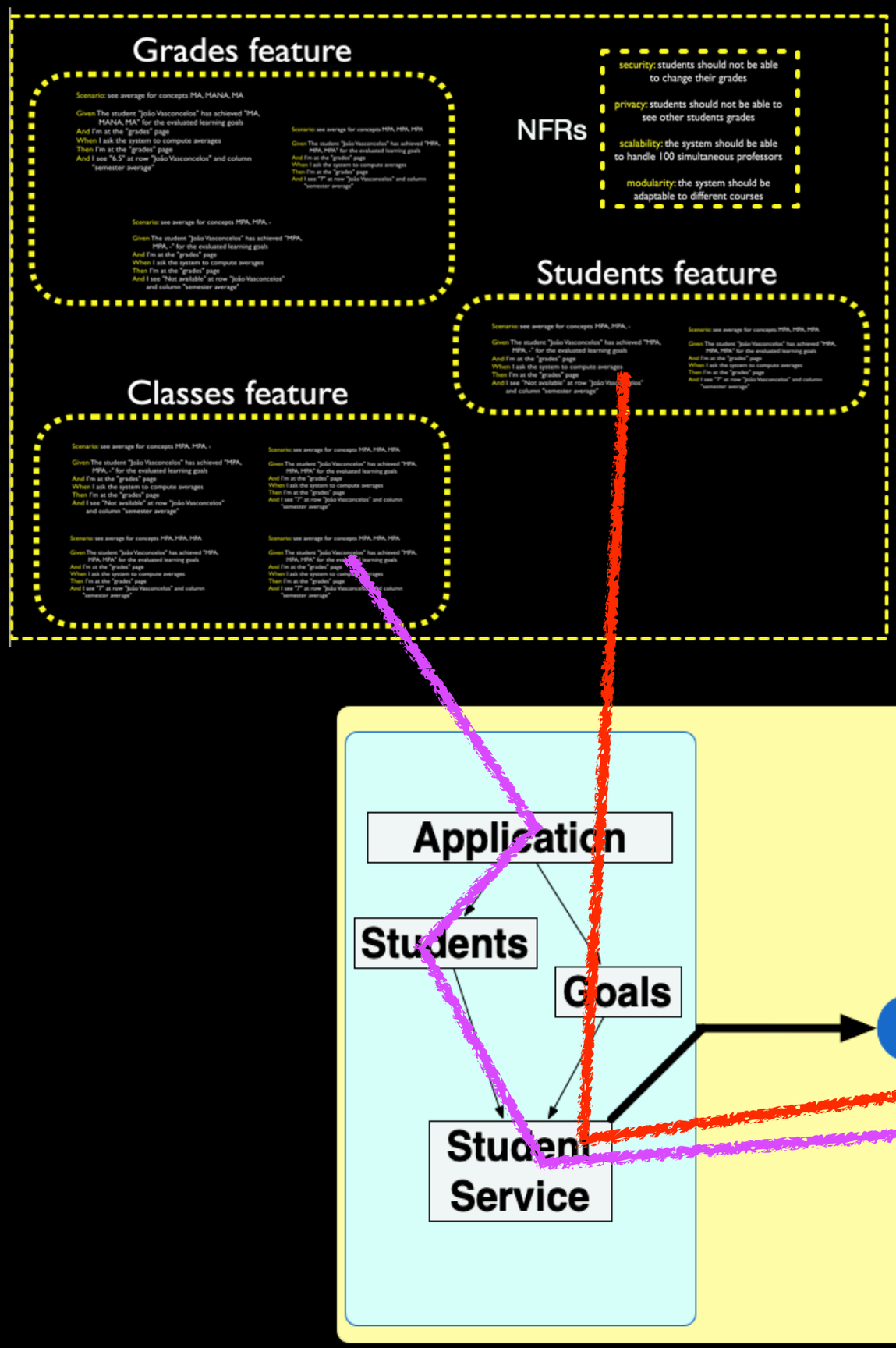
All driven by the (specified) system requirements

If there is a need to import student grades from a spreadsheet


StudentList
Import
StudentList
add(Student)
remove(Student)
*
Student
name
email
1
enrolments
parses
grades spreadsheet
and update
evaluations
Grades
Import
Service
Services
addStudent
removeStudent
synchronize
Classes
*
Class
semester
*
*
student
Enrolment
*
auto-evaluations
*
evaluations
*
Grade
Notification
EmailSender
send(String[])
Evaluation
goal
grade

Check if scenarios can be implemented by that





Mentally running scenario actions against model structure

To avoid misfits, aim at one-to-one mapping between purposes and concepts

Purposes, Concepts, Misfits, and a Redesign of Git

Santiago Perez De Rosso Daniel Jackson

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA, USA
{sperezde, dnj}@csail.mit.edu

Abstract

Git is a widely used version control system that is powerful but complicated. Its complexity may not be an inevitable consequence of its power but rather evidence of flaws in its design. To explore this hypothesis, we analyzed the design of Git using a theory that identifies concepts, purposes, and misfits. Some well-known difficulties with Git are described, and explained as misfits in which underlying concepts fail to meet their intended purpose. Based on this analysis, we designed a reworking of Git (called Gitless) that attempts to remedy these flaws.

To correlate misfits with issues reported by users, we conducted a study of Stack Overflow questions. And to determine whether users experienced fewer complications using Gitless in place of Git, we conducted a small user study. Results suggest our approach can be profitable in identifying

DDD models

- abstraction
- about information, stored and manipulated by the system, and its containment and association relations
- objects that create, store, and accesses such information
- not about GUI, pages, and navigation
- not about authorisation, responsibility, invocation, influence, executes, requests service, etc.

The system design captures a common language used by the team

The system design is distilled knowledge on how the team structures domain knowledge and understand main concepts

The system design summarises the implementation

System modelling studio

Hands on exercises

Design, implementation, and maintenance I

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Design, implementation, and maintenance 2

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Which principles guide the design process?

Design process should also be guided by…

Values

Principles

Patterns

architectural, design, implementation

Coding and design values

- Communication: focus on the reader
- Simplicity: minimize complexity dependencies, obscurity, over engineering
- Flexibility: easy to change, in expected ways
- Precision: names aligned to behavior

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Modularity principles

- Low coupling, reduced dependencies

  method calls

  object creation

  references

  inheritance, implementation, parametrisation

- High cohesion, focus on a single concern
- Hide information that is likely to change, or not useful for clients

Main motivation is to simplify and enable independent…

- understanding (local reasoning)
- maintenance (extensibility)
- development
- reuse

that's modularity!

avoids rippling effects

A modular system is...

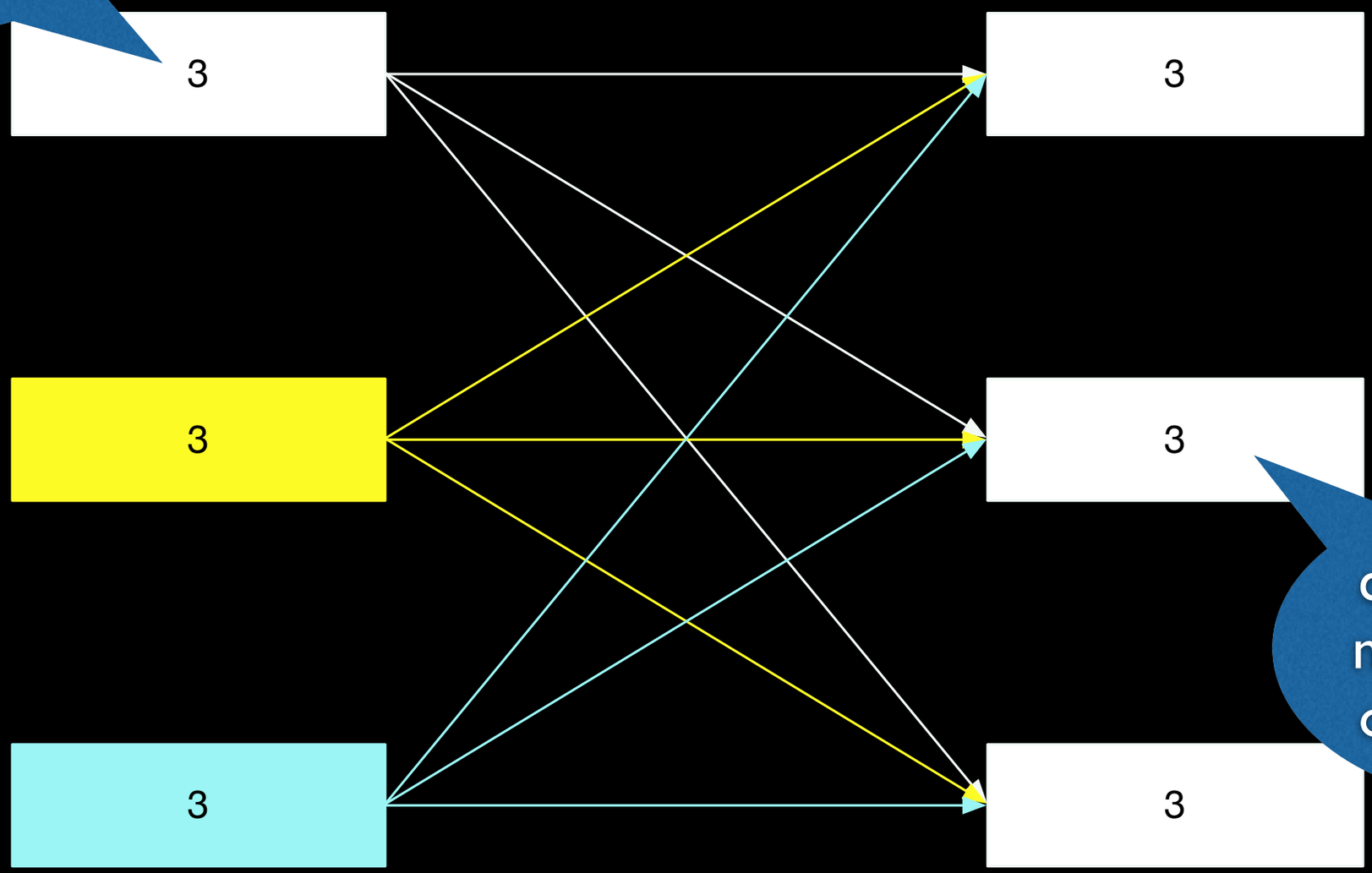
- split into separate units, language elements, modules
- developed by separate work assignments, design elements, modules

relative notion:
depends on what you intend to independently develop, maintain, and understand!

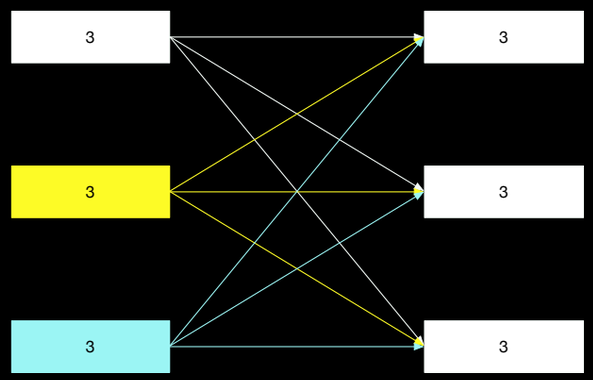
relative notion: depends on what more likely changes or varies!

High coupling, too many dependencies

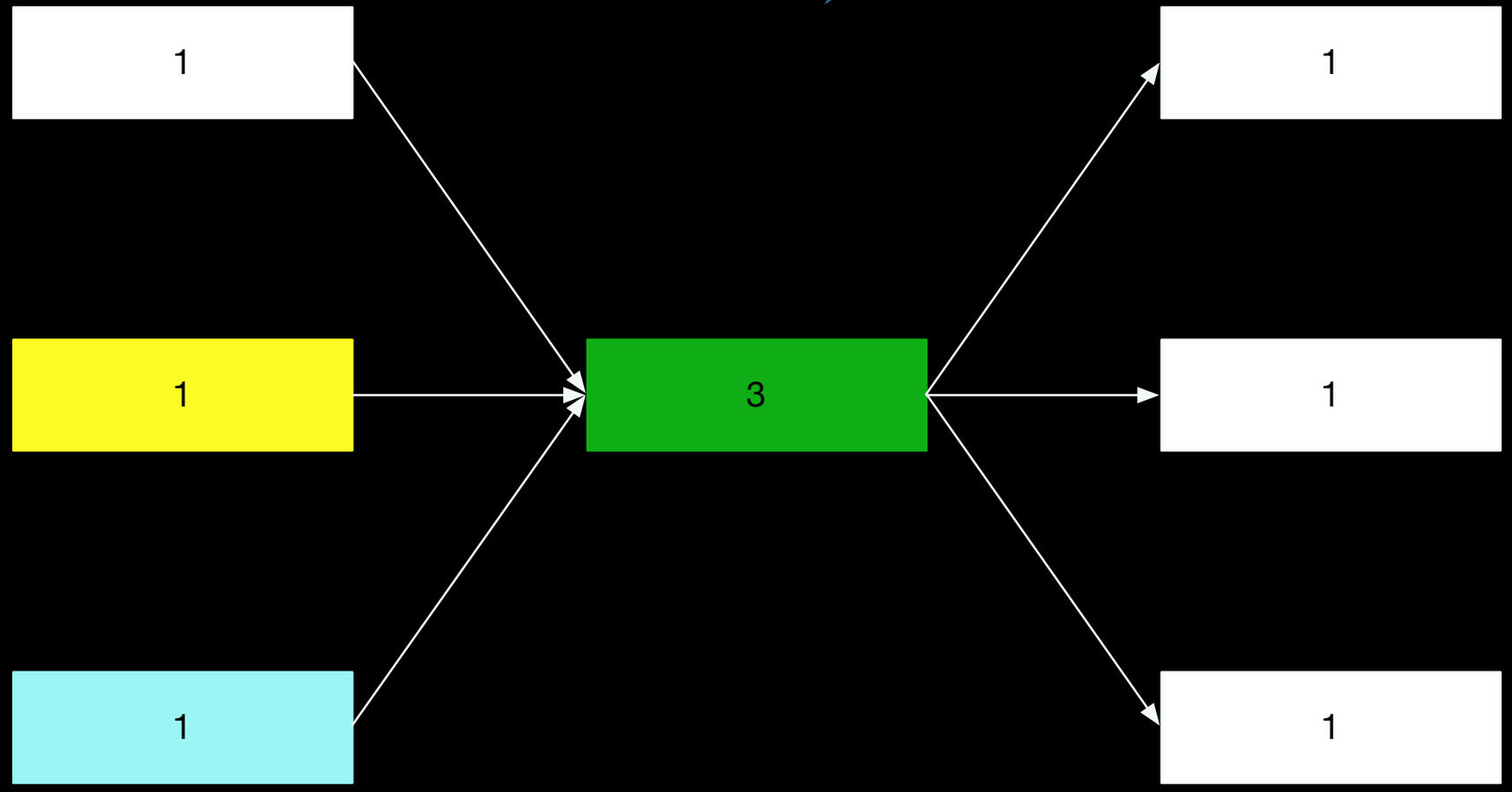
might be affected by changes in 3 other classes



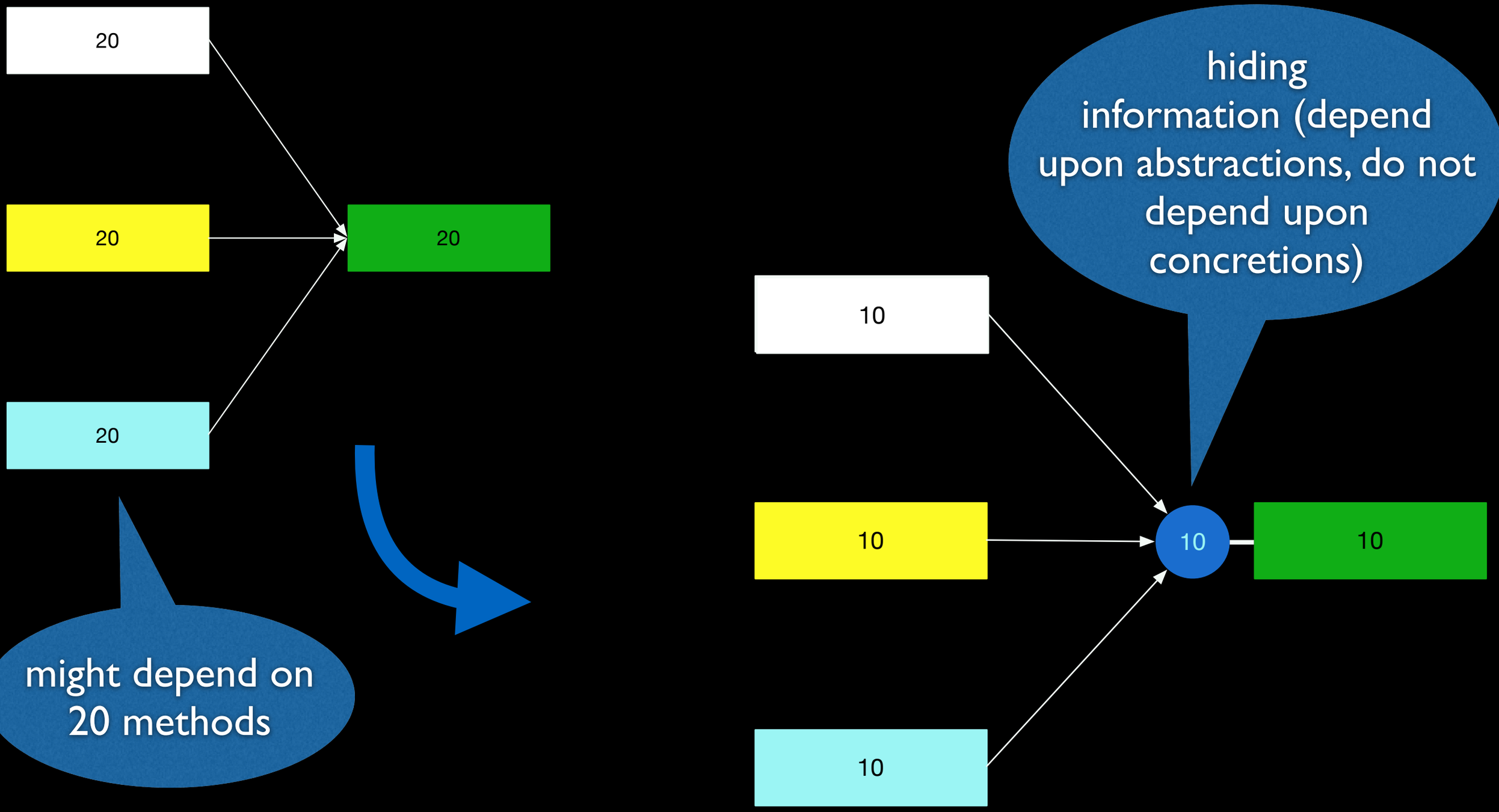
Low coupling: reducing dependencies by adding intermediate class



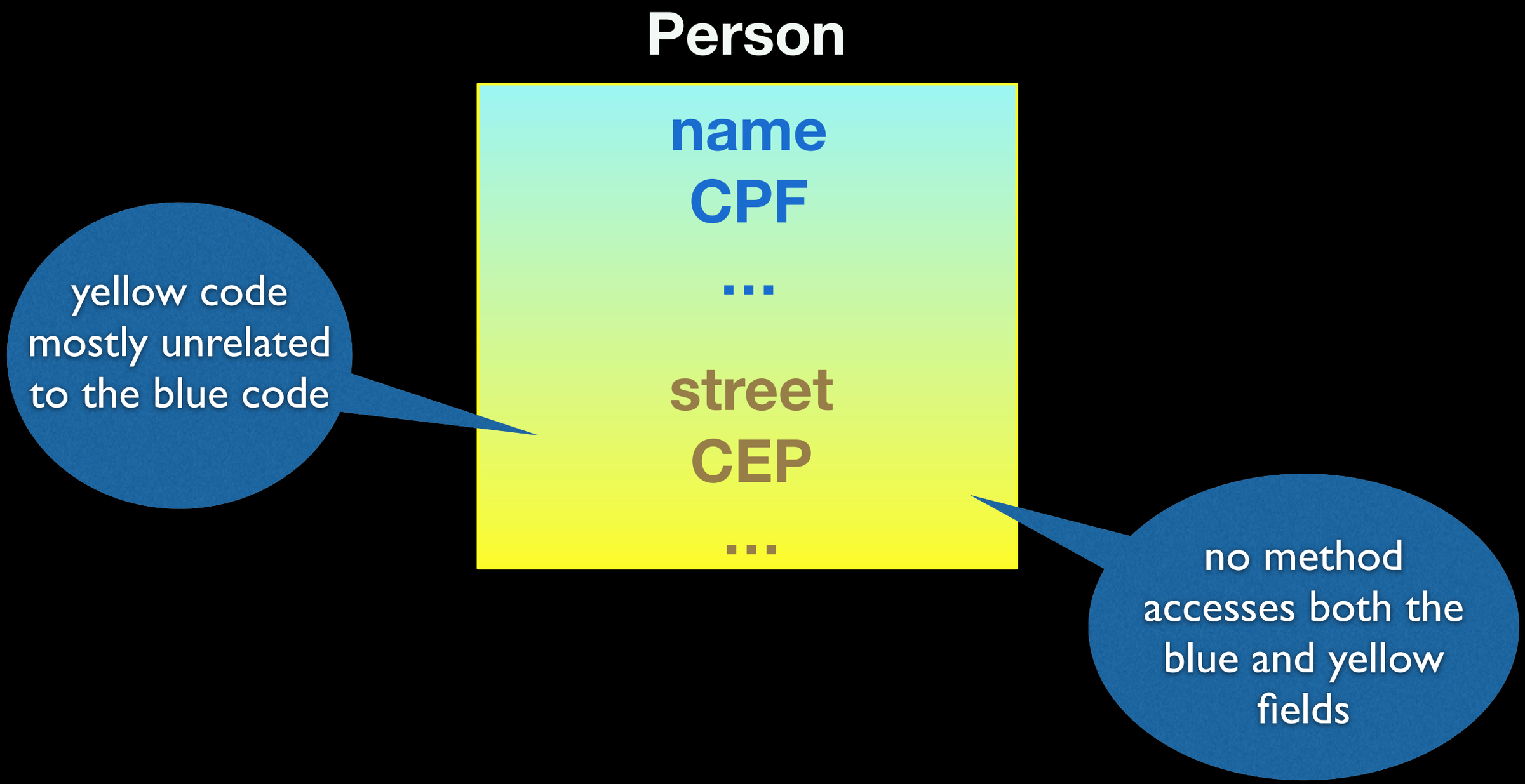
reduces complexity,
introduces indirection



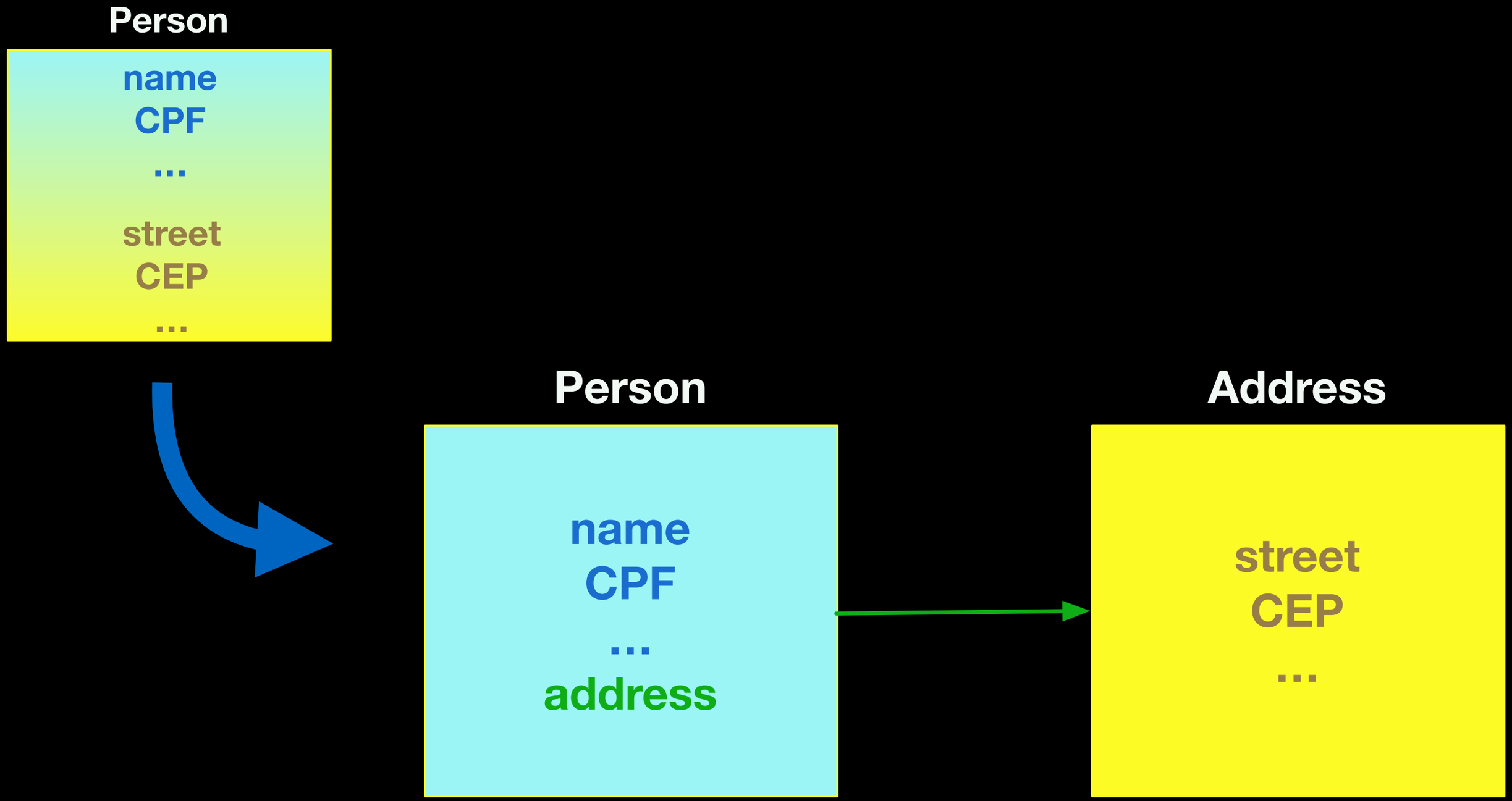
Low coupling: reducing dependencies by adding an interface



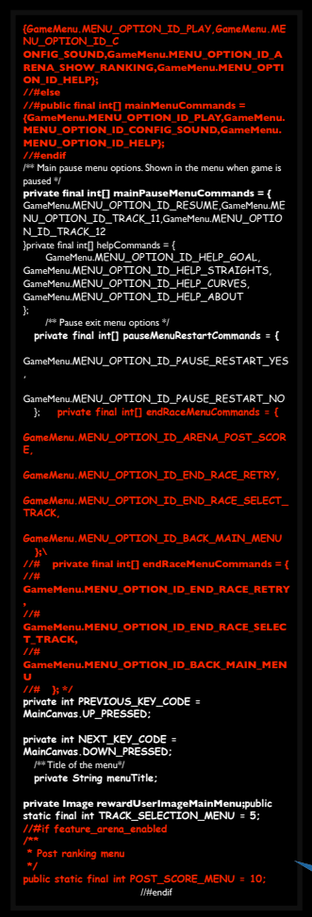
Low cohesion, mixing code with different concerns



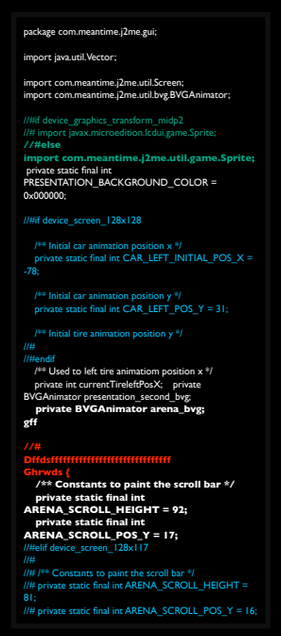
High cohesion: increasing focus by splitting a module



Low cohesion and agility, more defects, high costs



```
{GameMenu.MENU_OPTION_ID_PLAY,GameMenu.ME
NU_OPTION_ID_C
ONFIG_SOUND,GameMenu.MENU_OPTION_ID_A
RENA_SHOW_RANKING,GameMenu.MENU_OPTI
ON_ID_HELP};
//#else
//#public final int[] mainMenuCommands =
{GameMenu.MENU_OPTION_ID_PLAY,GameMenu.
MENU_OPTION_ID_CONFIG_SOUND,GameMenu.
MENU_OPTION_ID_HELP};
//#endif
/** Main pause menu options. Shown in the menu when game is
paused */
private final int[] mainPauseMenuCommands = {
GameMenu.MENU_OPTION_ID_RESUME,GameMenu.ME
NU_OPTION_ID_TRACK_11,GameMenu.MENU_OPTIO
N_ID_TRACK_12
}private final int[] helpCommands = {
        GameMenu.MENU_OPTION_ID_HELP_GOAL,
GameMenu.MENU_OPTION_ID_HELP_STRAIGHTS,
GameMenu.MENU_OPTION_ID_HELP_CURVES,
GameMenu.MENU_OPTION_ID_HELP_ABOUT
};
        /** Pause exit menu options */
    private final int[] pauseMenuRestartCommands = {

GameMenu.MENU_OPTION_ID_PAUSE_RESTART_YES
,
GameMenu.MENU_OPTION_ID_PAUSE_RESTART_NO
    };     private final int[] endRaceMenuCommands = {

GameMenu.MENU_OPTION_ID_ARENA_POST_SCOR
E,

GameMenu.MENU_OPTION_ID_END_RACE_RETRY,

GameMenu.MENU_OPTION_ID_END_RACE_SELECT_
TRACK,

GameMenu.MENU_OPTION_ID_BACK_MAIN_MENU
    };\
//#    private final int[] endRaceMenuCommands = {
//#
GameMenu.MENU_OPTION_ID_END_RACE_RETRY
,
//#
GameMenu.MENU_OPTION_ID_END_RACE_SELEC
T_TRACK,
//#
GameMenu.MENU_OPTION_ID_BACK_MAIN_MEN
U
//#    }; */
private int PREVIOUS_KEY_CODE =
MainCanvas.UP_PRESSED;

private int NEXT_KEY_CODE =
MainCanvas.DOWN_PRESSED;
    /** Title of the menu*/
    private String menuTitle;

private Image rewardUserImageMainMenu;public
static final int TRACK_SELECTION_MENU = 5;
//#if feature_arena_enabled
/**
 * Post ranking menu
 */
public static final int POST_SCORE_MENU = 10;
                                //#endif
```



```
package com.meantime.j2me.gui;

import java.util.Vector;

import com.meantime.j2me.util.Screen;
import com.meantime.j2me.util.bvg.BVGAnimator;

//#if device_graphics_transform_midp2
//# import javax.microedition.lcdui.game.Sprite;
//#else
import com.meantime.j2me.util.game.Sprite;
//#endif
 private static final int
PRESENTATION_BACKGROUND_COLOR =
0x000000;

//#if device_screen_128x128

    /** Initial car animation position x */
    private static final int CAR_LEFT_INITIAL_POS_X =
-78;

    /** Initial car animation position y */
    private static final int CAR_LEFT_POS_Y = 31;

    /** Initial tire animation position y */
//#
//#endif
    /** Used to left tire animatiom position x */
    private int currentTireleftPosX;    private
BVGAnimator presentation_second_bvg;
    private BVGAnimator arena_bvg;
gff

//#
Dffdsfffffffffffffffffffffffffffffffff
Ghrwds {
    /** Constants to paint the scroll bar */
    private static final int
ARENA_SCROLL_HEIGHT = 92;
    private static final int
ARENA_SCROLL_POS_Y = 17;
//#elif device_screen_128x117
//#
/// /** Constants to paint the scroll bar */
/// private static final int ARENA_SCROLL_HEIGHT =
81;
/// private static final int ARENA_SCROLL_POS_Y = 16;
```

```
//#if sku_id_sel
//#
//#g = originalG;
//#g.drawImage(bufferImg, 0, 0, 0);
//#
//#endif} catch(Exception e){
e.printStackTrace();
}
}
public void changeScreen (innewScreenIndex;
this.nextScreenMode = newScreenMode;
this.nextScreenParam = param;    /**
     * Shows the last screen shown.
     * If there's not a last screen (i.e., the screen index in
     * the stack is public void goBack() {
     int[] screenData = this.popSc
       // avoid mainCanvas to go back to a waiting server
screen
       if (screenData[STACK_SCREEN_MODE_INDEX] ==
Resources.MAIN_SCREEN_MODE_ARENA_WAITING_
SERVER) {
           this.goBack();

return;this.changeScreen(screenData[STACK_SCREEN
_INDEX],
  screenData[STACK_SCREEN_MODE_INDEX],
  screenData[STACK_PARAM_INDEX]);
       }
this.goBack = true;
    }//#public void commandAction(Command c, Displayable d) {
//#if (c == this.leftCommand) {
//#this.keyPressed(LEFT_SOFT_KEY);
//#} else if (c == this.rightCommand) {
//#this.keyPressed(RIGHT_SOFT_KEY);
//#}
//#}
//#
//#endif
```

different colors represent code associated with different features, or concerns

tangling: mixing different concerns in the same module

scattering: concern implementation not in a single module

Modularity through high cohesion and low coupling

strongly connected, strongly related, inside

weakly connected, outside

Software and systems engineering

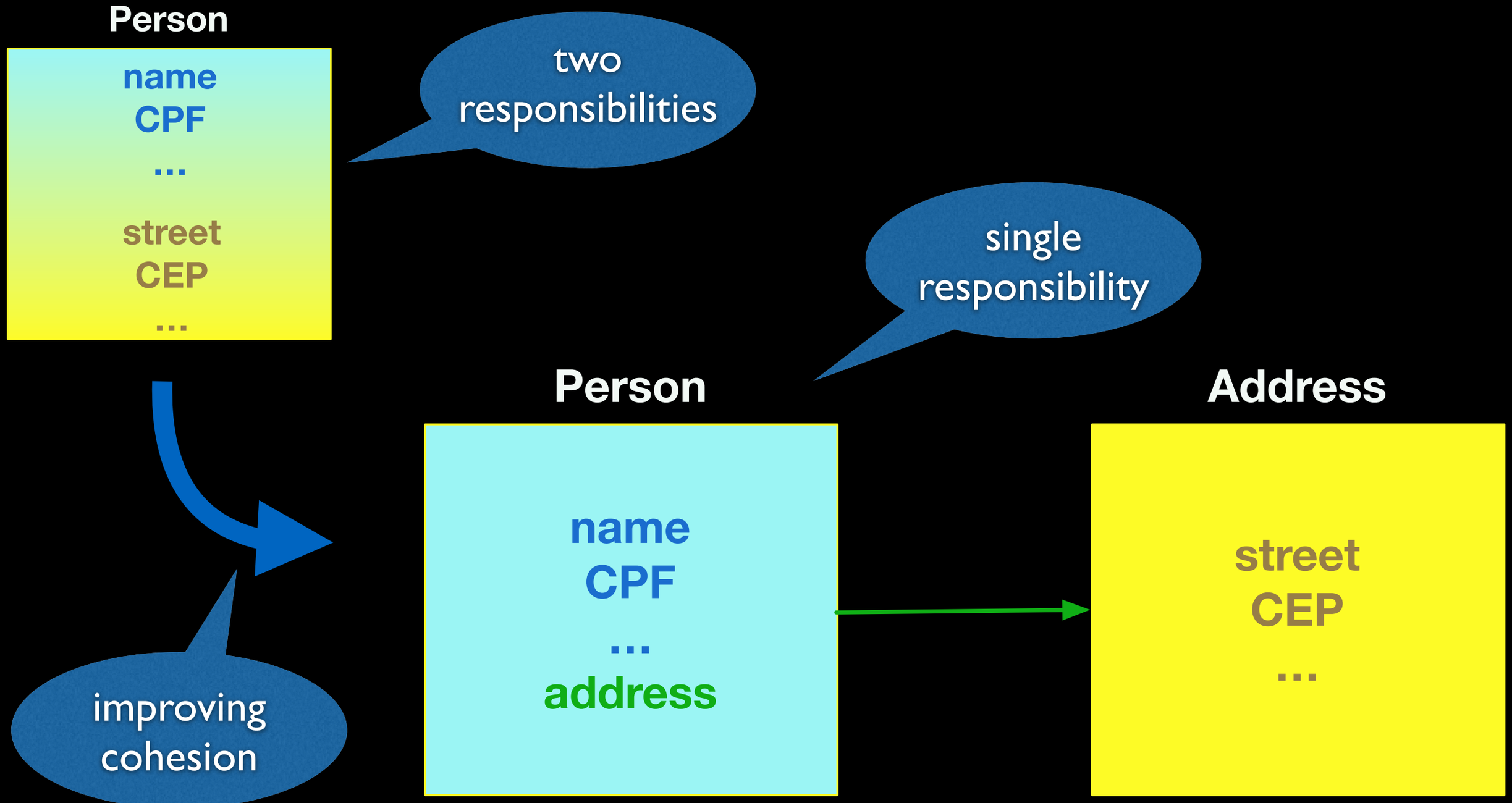
Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

SOLID design principles

- Single responsibility
- Open-closed
- Liskov Substitution
- Interface segregation
- Dependency injection
- Demeter

Single responsibility: a class should have a single responsibility



Open-closed: a module should be open for extension but closed for modification

```
class Account {
   double balance;

   void credit(double v) {
     balance = balance + v;
   }
   void debit(double v) {
     balance = balance - v;
   }
   …
}
```

```
class SpecialAccount extends Account {
   double bonus;

   void credit(double v) {
     super.credit(v);
     bonus = bonus + (0.01 * v);
   }
   …
}
```

open for extension

```
a = new Account();
a.credit(576.35);
```

closed for modification

Extension requires modification: no use of sub typing

```
class Account {
   double balance;
   String type;
   …
   void credit(double v) {
     if (type.equals(“Basic”)) {
       balance = balance + v;
     } else {
        …
     }
   }

   …
}
```

handling a new account type requires a new conditional

Liskov substitution: objects of subclasses should behave like those of superclasses if used via superclasses methods

```
class Account {
   double balance;

   void credit(double v) {
     balance = balance + v;
   }
   void debit(double v) {
     balance = balance - v;
   }
   …
}
```

clients can observe that objects of this class behave differently from objects of its superclass

```
class AccountCPMF extends Account {
   void debit(double v) {
     balance = balance - (1.01 * v);
   }
   …
}
```

call debit, and then print balance and you can observe the difference

breaks LSP!

Breaking LSP reduces extensibility

```
double m(Account a) {
   a.credit(v);
   …
   a.debit(v);
   return a.getBalance();
}
```

```
double m(Account a) {
   …
   return a.getBalance();
}
```

safe, behavior-preserving modification, based only on the code of Account, if LSP is followed

possibly unsafe, non behavior-preserving modification, if LSP is not followed (demands analysis of all subclasses of Account)

Supporting LSP by moving incompatible behavior from superclass to sibling

```
abstract class Account {
   double balance;

   void credit(double v) {
     balance = balance + v;
   }
   void debit(double v);
   …
}
```

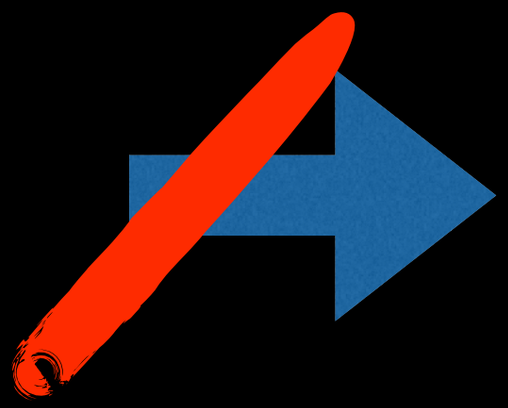
```
class StandardAccount extends Account {
   void debit(double v) {
     balance = balance - v;
   }
   …
}
```

```
class AccountCPMF extends Account {
   void debit(double v) {
     balance = balance - (1.01 * v);
   }
   …
}
```

No misleading reasoning based on abstract methods

```
double m(Account a) {
   a.credit(v);
   …
   a.debit(v);
   return a.getBalance();
}
```

```
double m(Account a) {
   …
   return a.getBalance();
}
```



abstract method declaration specifies that we can’t expect a particular method behavior

LSP doesn't prevent a subclass from doing more (additional methods, fields, and behavior on those fields)

```
class Account {
   double balance;

   void credit(double v) {
     balance = balance + v;
   }
   void debit(double v) {
     balance = balance - v;
   }
   …
}
```

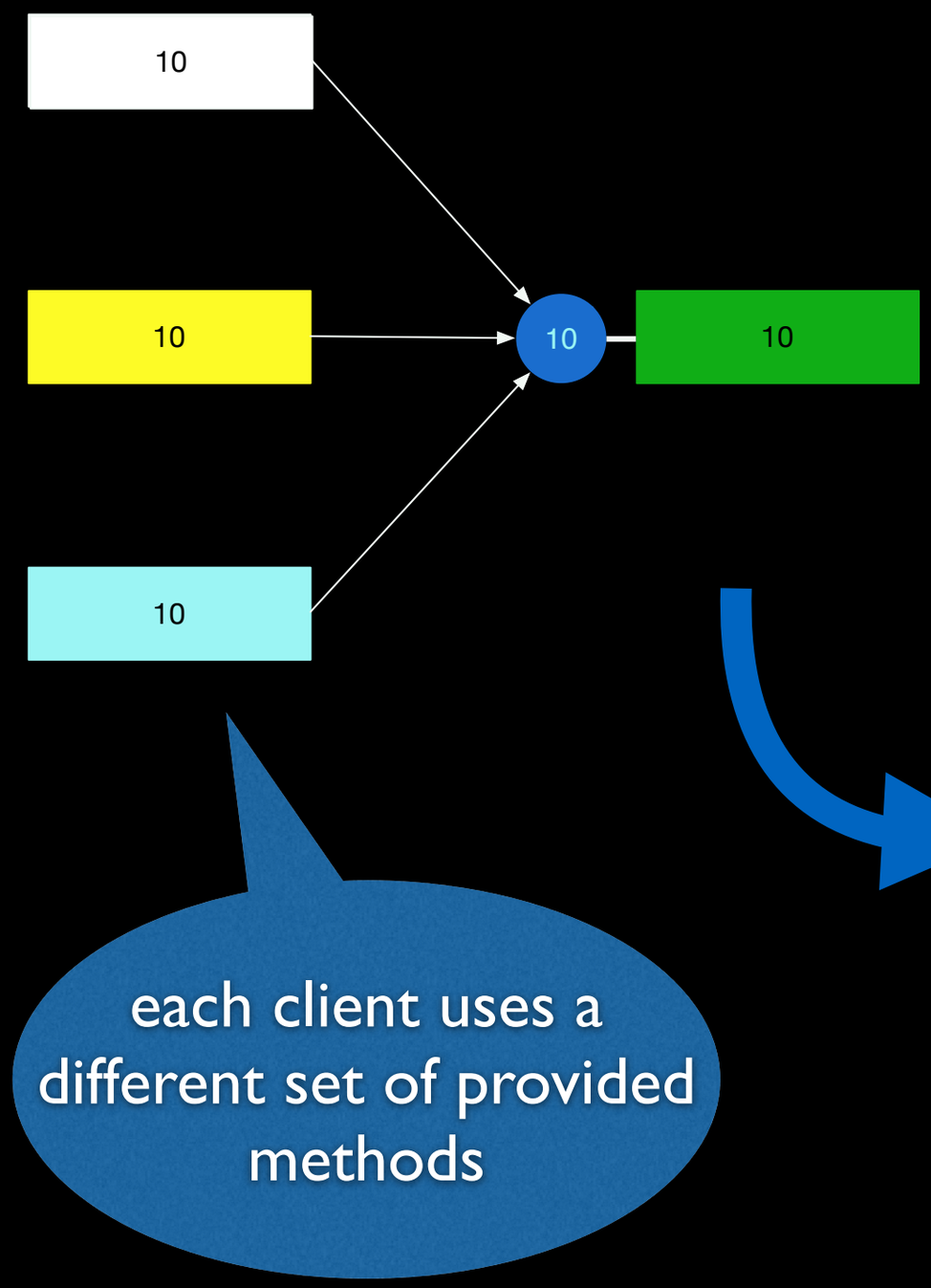
call credit, and then print balance and you can't observe the difference

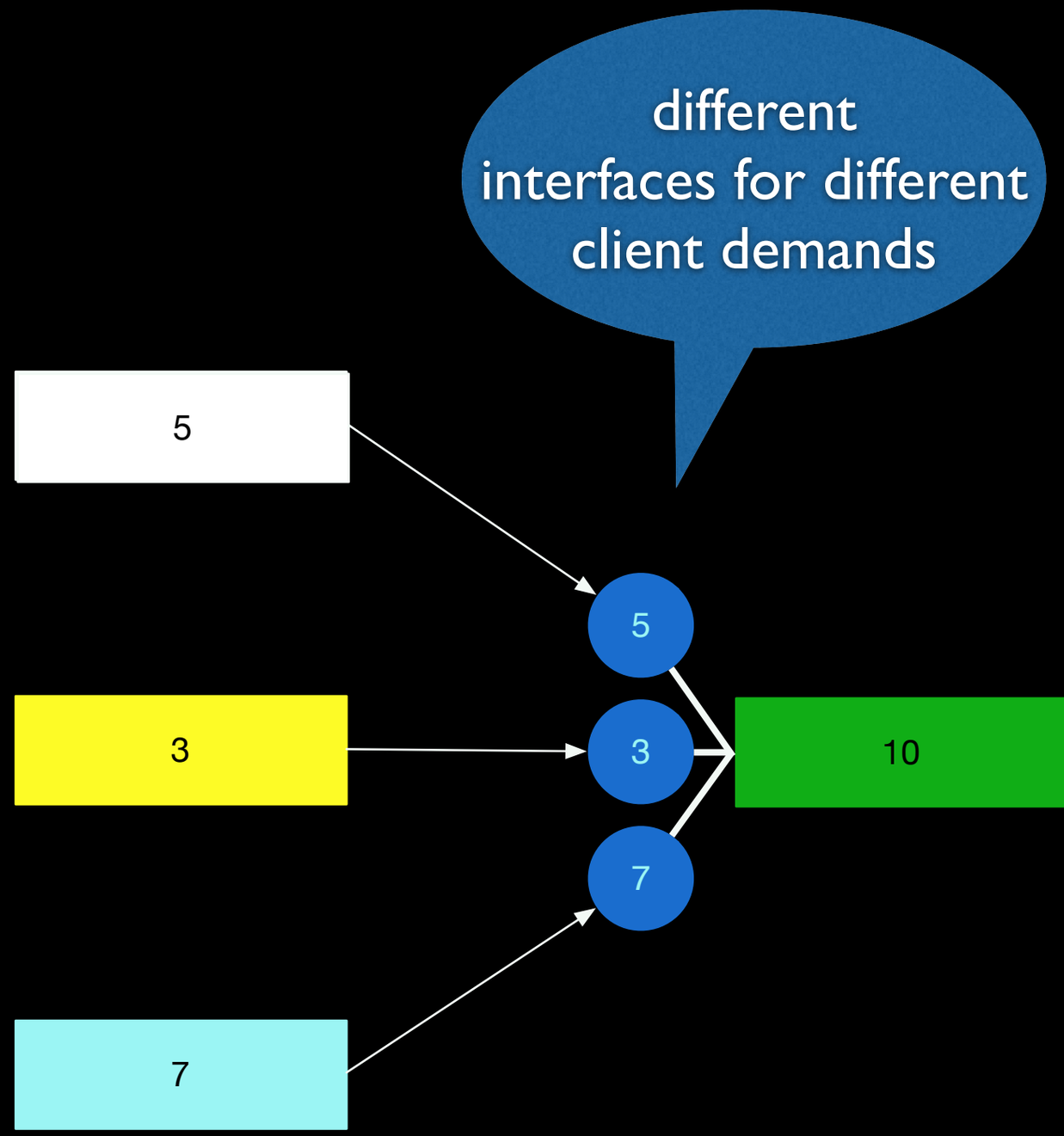
clients can only observe that objects of this class behave differently from objects of its superclass when getting bonus

```
class SpecialAccount extends Account {
   double bonus;

   void credit(double v) {
     super.credit(v);
     bonus = bonus + (0.01 * v);
   }
   …
}
```

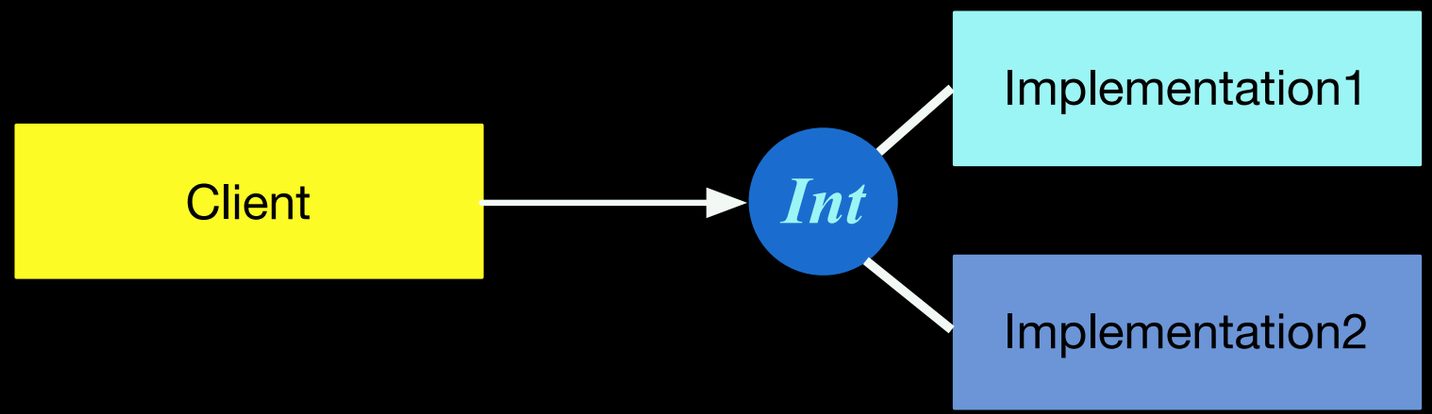
satisfies LSP!

Interface segregation: client specific interfaces are better than one general purpose interface





Dependency injection: dependencies should be injected into a dependent object




```
class Client {
   Int int;

   Client() {
     int = new Implementation1();
   }
   …
}
```

```
c = new Client();
c.m();
c.n();
```

internally
created dependency

Injected dependency

```
class Client {
    Int int;

    void setInt(Int i) {
      int = i;
    }
    …
}
```

```
c = new Client();
c.setInt(new Implementation1());
c.m();
c.n();
```

externally created dependency

```
c = new Client();
c.setInt(new Implementation2());
c.m();
c.n();
```

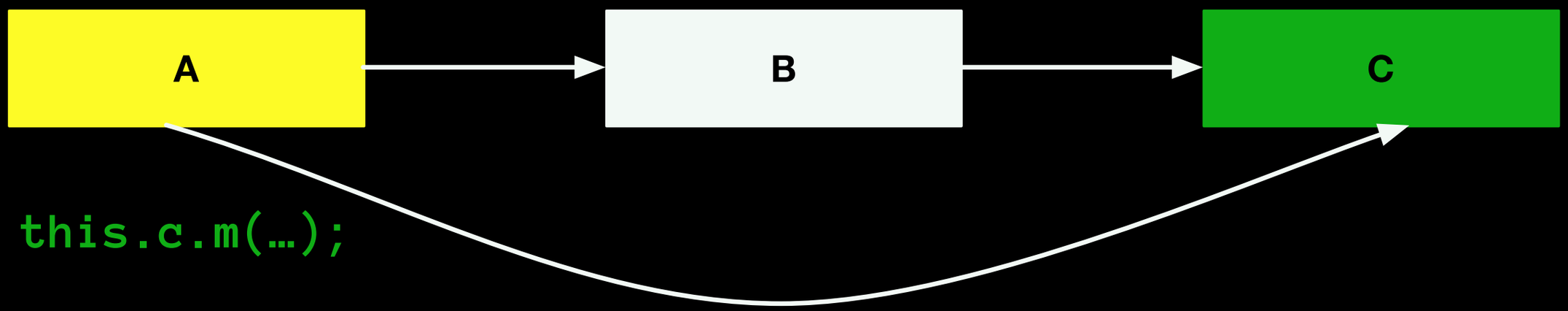
working with alternative dependency

```
c = new Client();
c.setInt(new Implementation1());
c.m();
…
c.setInt(new Implementation2());
c.n();
```

changing dependency at run time

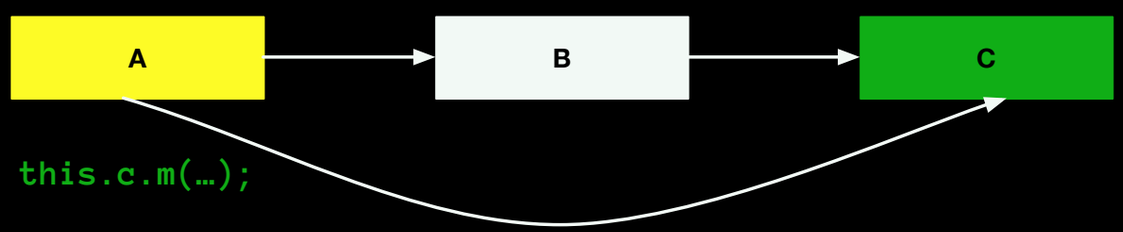
Demeter: a method of class A should only call methods of A or of the classes of A fields

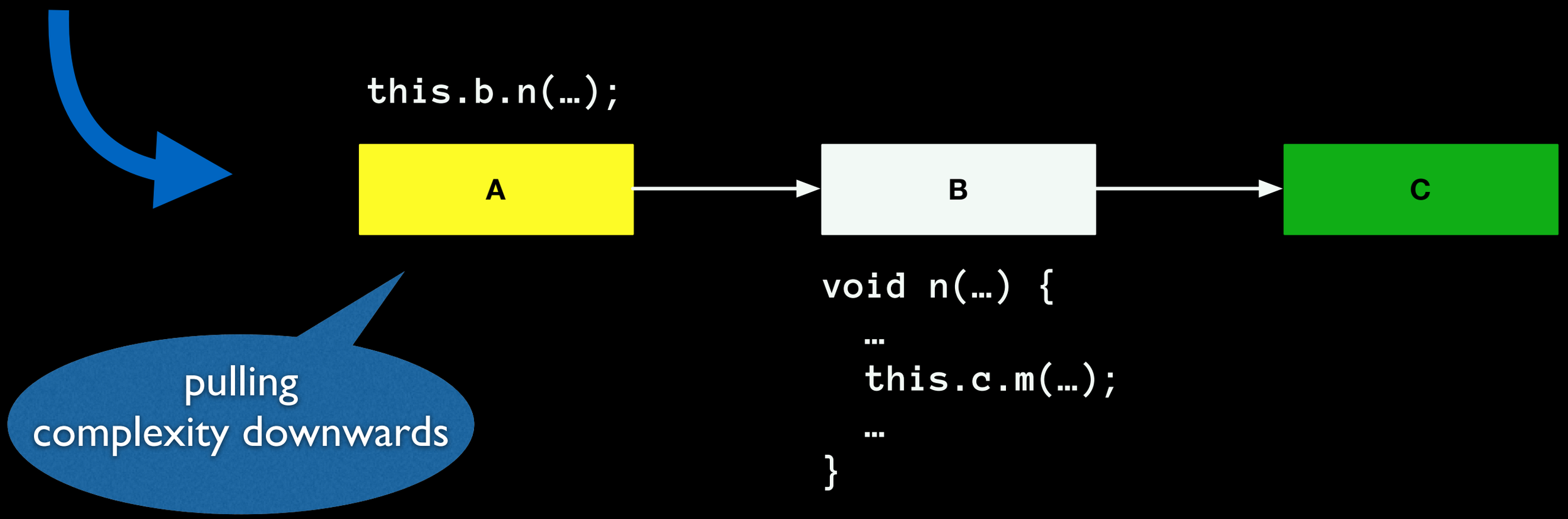
```
this.b.getC().m(…);
```



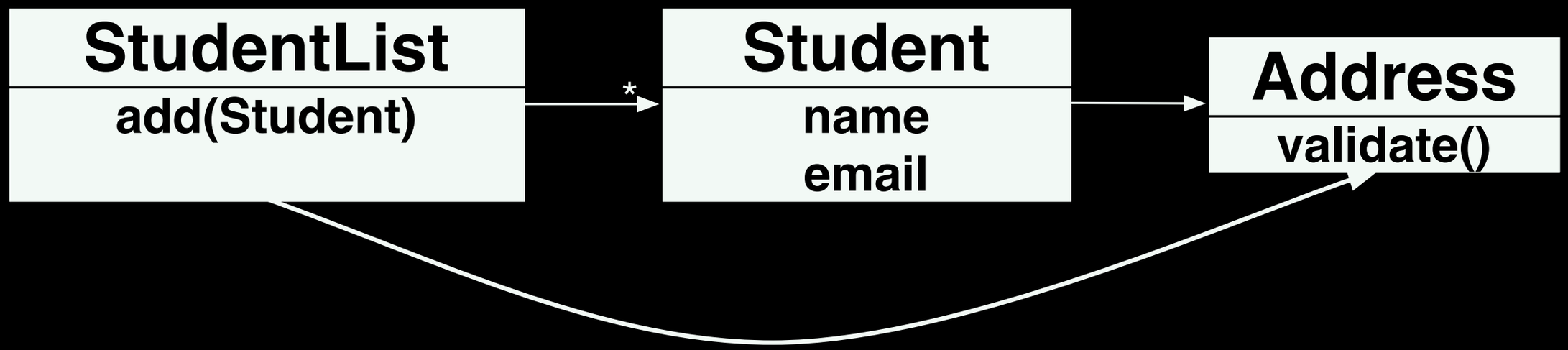
Reducing dependencies, pushing them down

```
this.b.getC().m(…);
```



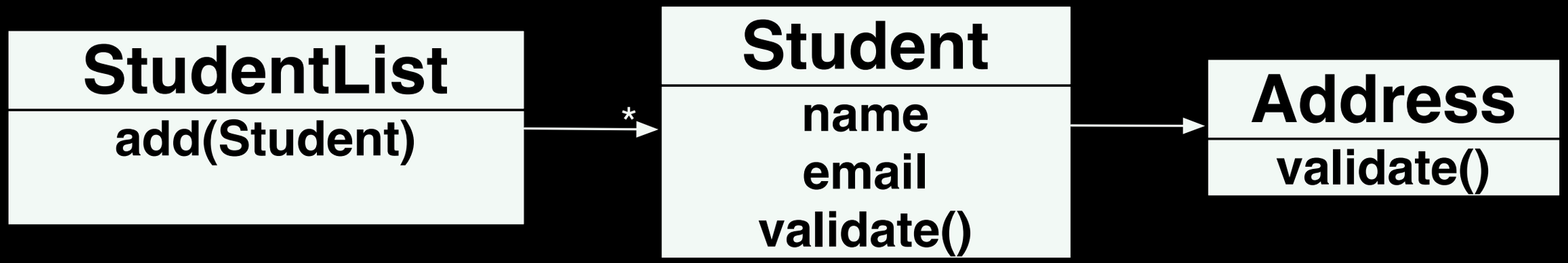


```
student.getAddress().validate();
```



breaks Demeter

```
student.validate();
```




```
this.address.validate();
```

satisfies Demeter

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Other design principles and values

Package cohesion principles

- Classes that change together, belong together (common closure)
- Classes that aren't reused together should not be grouped together (common reuse)
- The granule of reuse is the granule of release (release reuse equivalency)

app
controllers
articles_controller.rb
posts_controller.rb
models
article.rb
post.rb
views
article
posts
_table.html.erb
archived.html.erb
edit.html.erb
index.html.erb
show.html.erb

packages

post slice

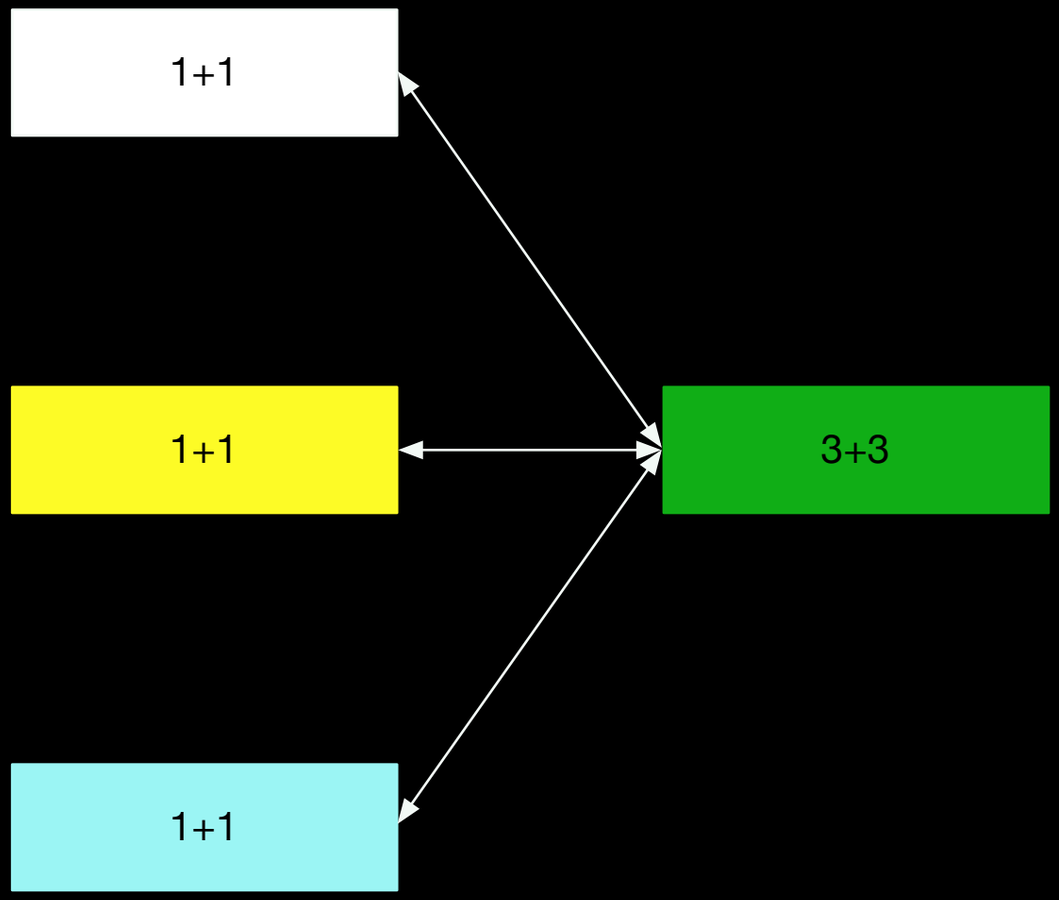
module structure established by the kinds of classes

module structure established by features and tasks

Package coupling principles

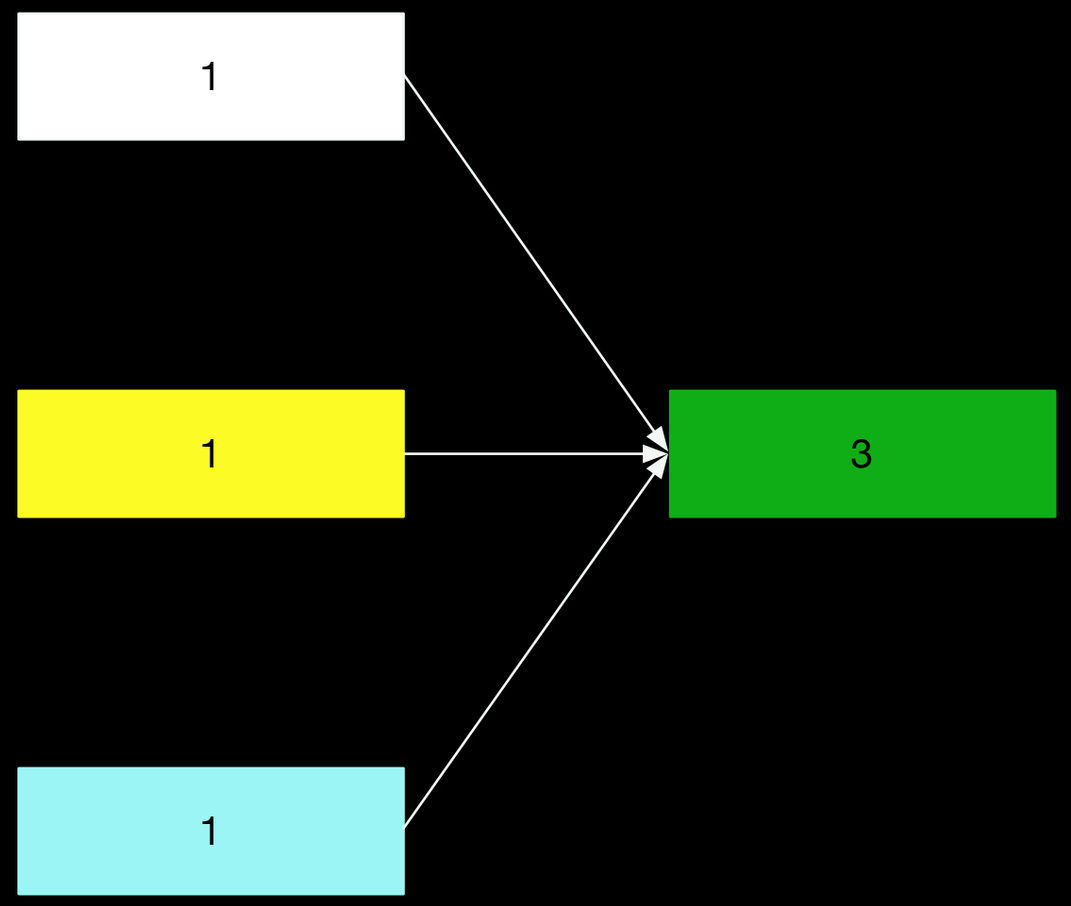
- The dependencies betwen packages must not form cycles (acyclic dependencies)
- Depend in the direction of stability (stable dependencies)
- Stable packages should be abstract packages (stable abstractions)

with interfaces or abstract classes

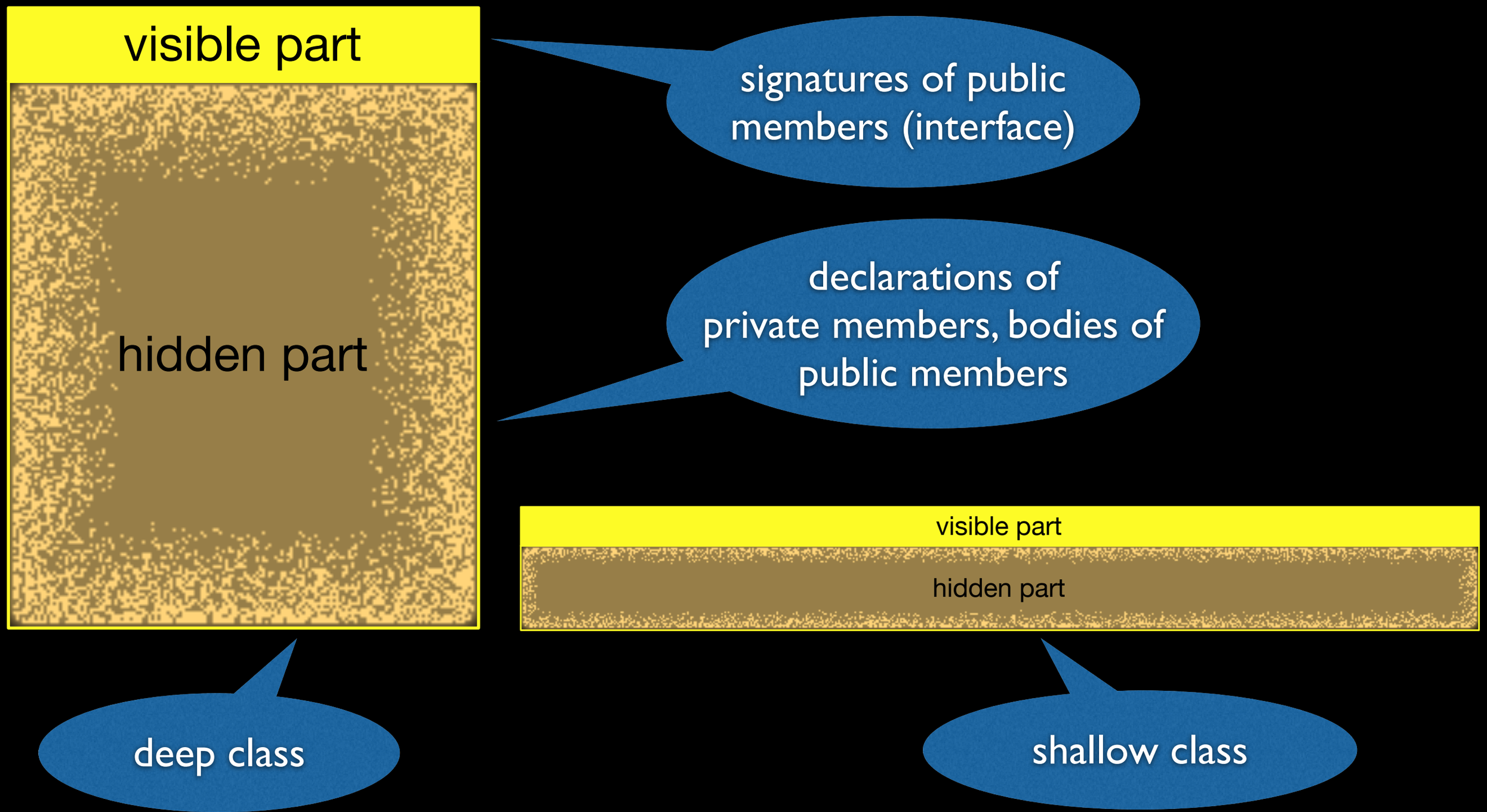


alternative solution colapses packages into a single one

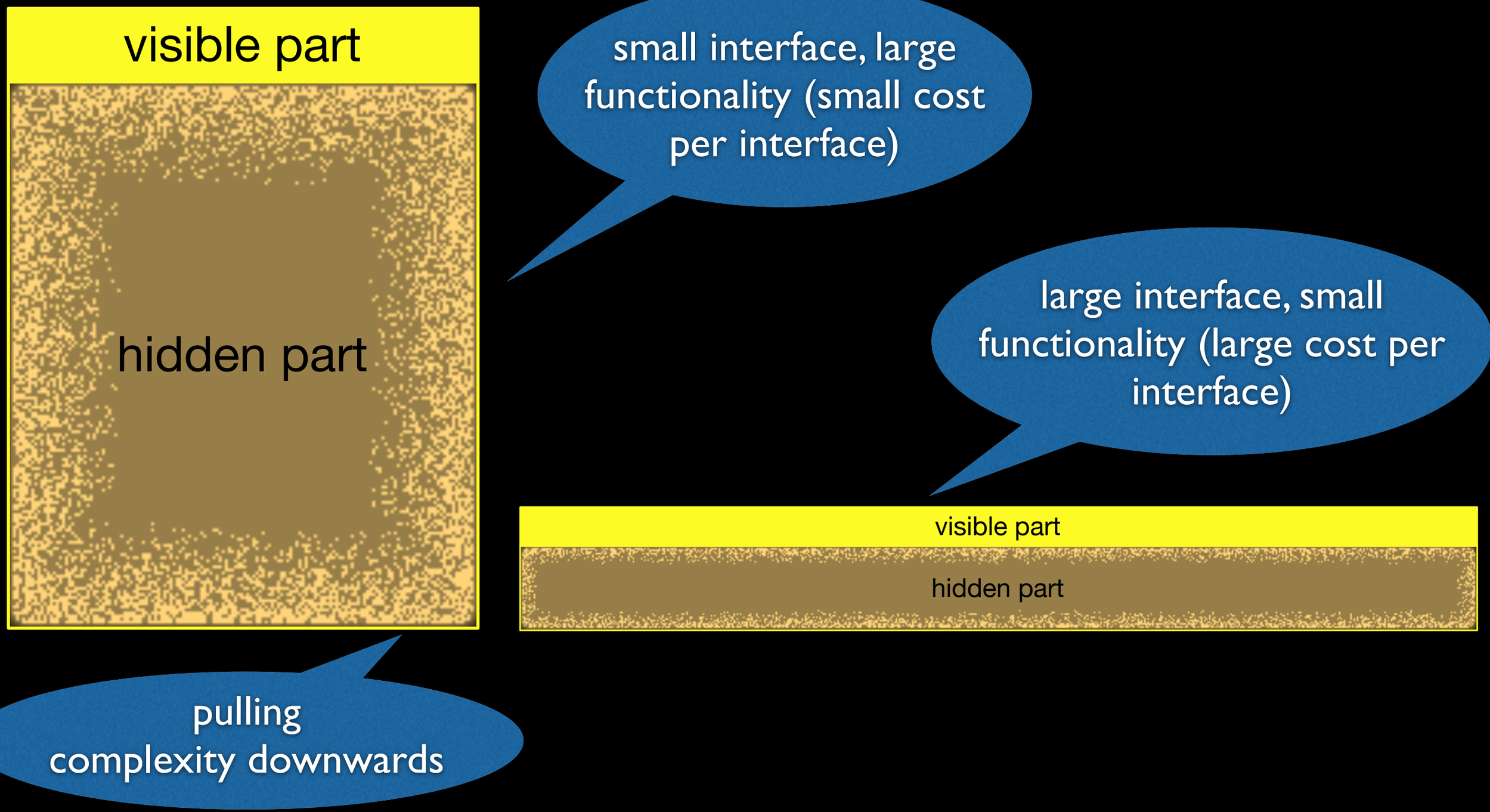
reducing package coupling by removing cyclic dependencies



Classes (modules) should be deep



Cost per interface



Abstraction is more important than size

- design and code should rely on abstraction
- large methods and classes are often associated with lack of abstraction
- so the emphasis on small methods and classes
- but what matters most is that methods and classes properly apply abstraction

Direction of control drives dependence direction (for reusable code)

- APIs, libraries, auxiliary classes: custom code calls reusable code
- Frameworks: reusable code calls custom code (inversion of control)

Other rules

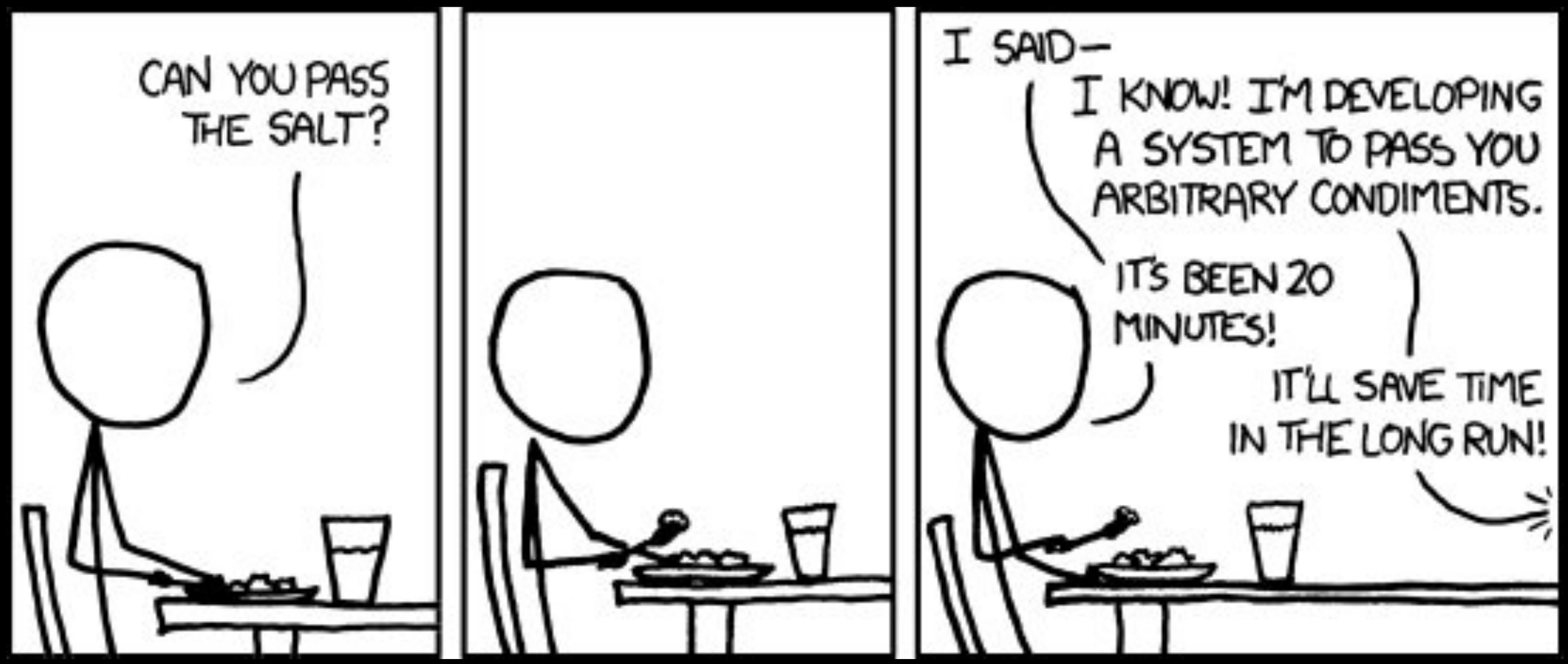
Rule of Modularity: Write simple parts connected by clean interfaces.
Rule of Clarity: Clarity is better than cleverness.
Rule of Composition: Design programs to be connected to other programs.
Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
Rule of Simplicity: Design for simplicity; add complexity only where you must.
Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
Rule of Transparency: Design for visibility to make inspection and debugging easier.
Rule of Robustness: Robustness is the child of transparency and simplicity.
Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.

The Art of Unix Programming, Eric Raymond

Other rules

Rule of Least Surprise: In interface design, always do the least surprising thing.
Rule of Silence: When a program has nothing surprising to say, it should say nothing.
Rule of Repair: When you must fail, fail noisily and as soon as possible.
Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
Rule of Diversity: Distrust all claims for “one true way”.
Rule of Extensibility: Design for the future, because it will be here sooner than you think.

The Art of Unix Programming, Eric Raymond



https://xkcd.com/

Hands on exercises

Design, implementation, and maintenance 2

Design, implementation, and maintenance 3

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Values and principles provide general direction, independent of specific problems

Patterns provide common solution structure for specific recurrent problems…

often applying values and principles

DDD already introduces a number of higher level modelling patterns

(entity, value object, repository, factory, service, etc.)

Pattern description

- Context (situation giving rise to a problem)
- Problem (recurring in such a context)
- Forces (requirements for the solution)
- Solution (proven, includes structure and dynamics)
- Consequences (after applying the pattern)
- Known uses (of the proven solution)

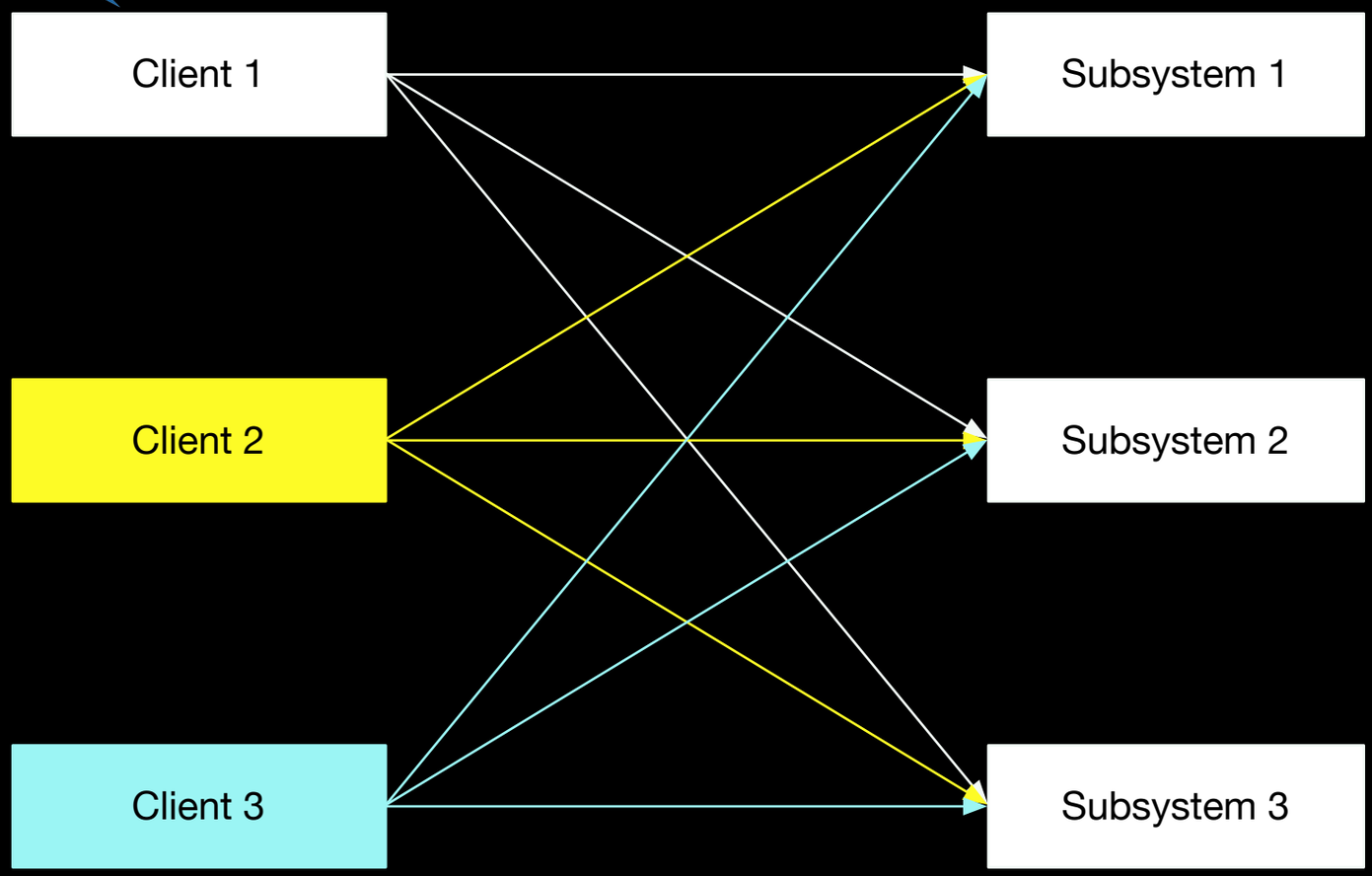
Facade design pattern

Context

system, library, or framework
with a complex set of classes

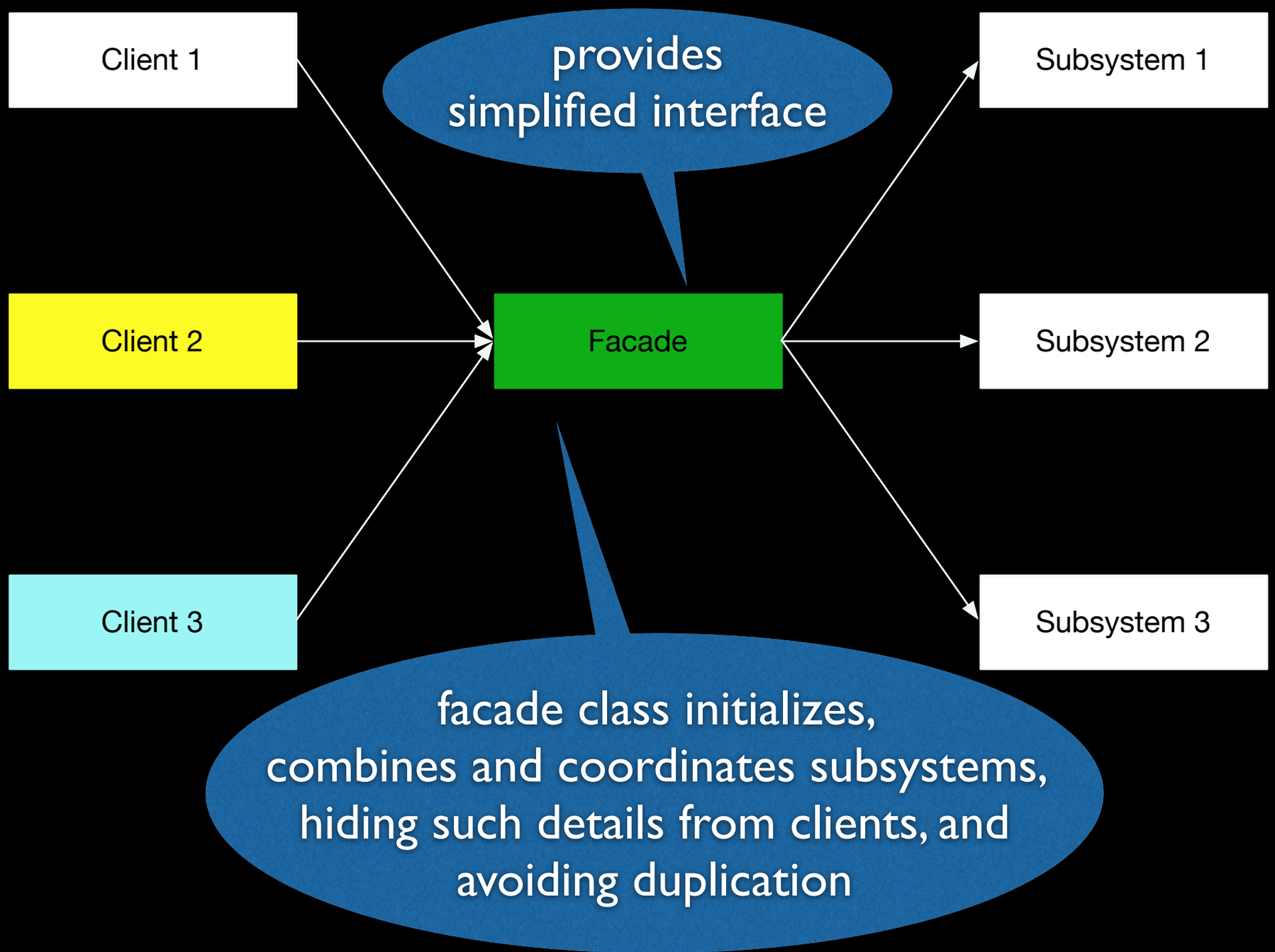
Problem

client tightly coupled to the subsystems



client has to know how to initialize, combine and coordinate subsystems

Solution (structure)



Analysis Patterns
Fowler
Addison Wesley

A Martin Fowler Signature Book
SERVICE DESIGN PATTERNS
DAIGNEAU
Addison Wesley

A Martin Fowler Signature Book
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
FOWLER
Addison Wesley

Schumacher
Fernandez-Buglioni
Hybertson
Buschmann
Sommerlad
SECURITY PATTERNS
Integrating Security and Systems Engineering
WILEY

Buschmann
Henney
Schmidt
Volume 5
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
WILEY

Buschmann
Henney
Schmidt
Volume 4
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
WILEY

Kircher
Jain
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Vol 3
WILEY

Schmidt
Stal
Rohnert
Buschmann
Volume 2
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
WILEY

Buschmann
Meunier
Rohnert
Sommerlad
Stal
PATTERN - ORIENTED SOFTWARE ARCHITECTURE
A SYSTEM OF PATTERNS
WILEY

Design Patterns
Gamma • Helm
Johnson • Vlissides
Addison Wesley

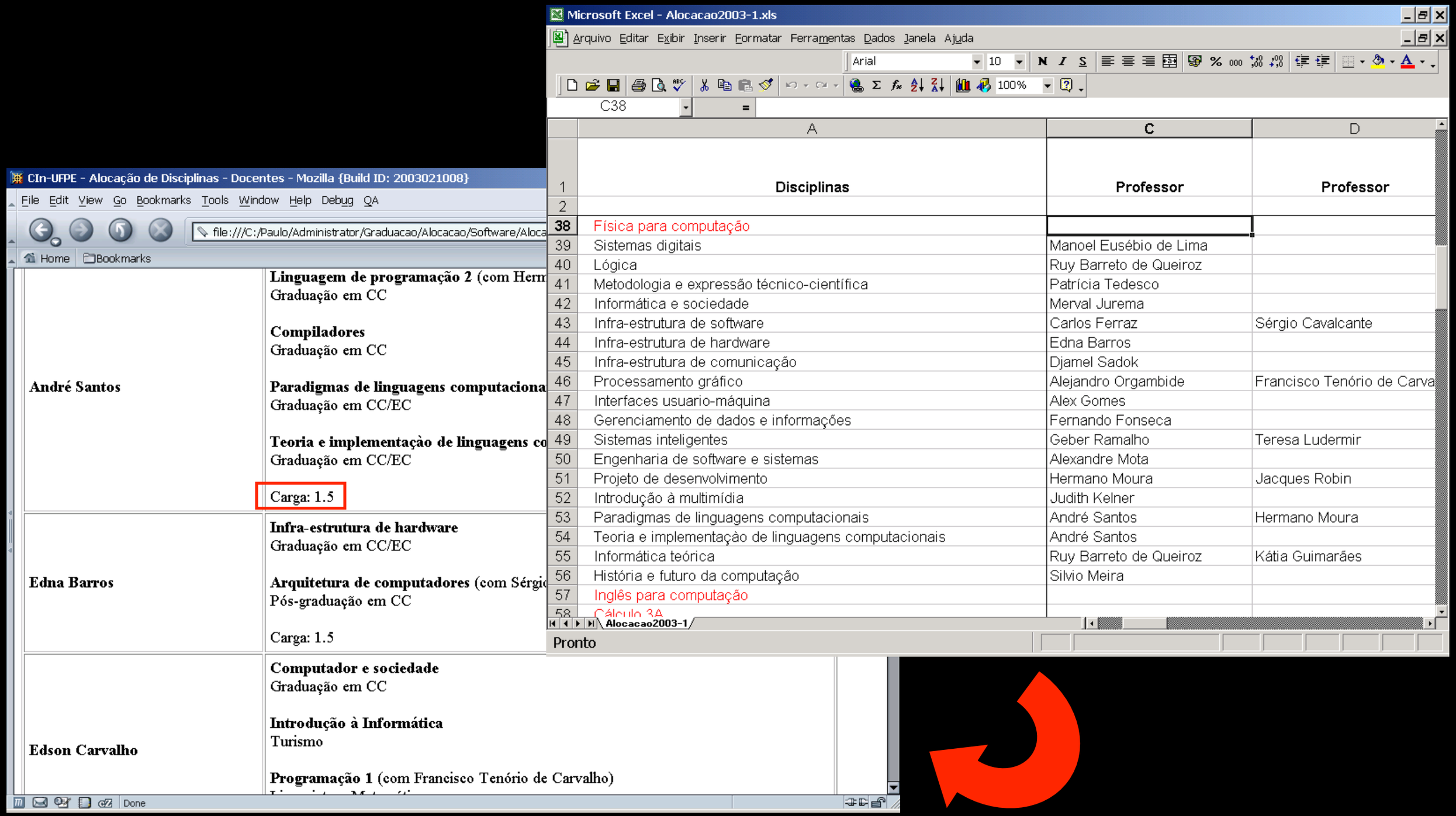
Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Applying the abstract factory design pattern

Professors and courses



	A	C	D
1	Disciplinas	Professor	Professor
2			
38	Física para computação		
39	Sistemas digitais	Manoel Eusébio de Lima	
40	Lógica	Ruy Barreto de Queiroz	
41	Metodologia e expressão técnico-científica	Patrícia Tedesco	
42	Informática e sociedade	Merval Jurema	
43	Infra-estrutura de software	Carlos Ferraz	Sérgio Cavalcante
44	Infra-estrutura de hardware	Edna Barros	
45	Infra-estrutura de comunicação	Djamel Sadok	
46	Processamento gráfico	Alejandro Orgambide	Francisco Tenório de Carva
47	Interfaces usuario-máquina	Alex Gomes	
48	Gerenciamento de dados e informações	Fernando Fonseca	
49	Sistemas inteligentes	Geber Ramalho	Teresa Ludermir
50	Engenharia de software e sistemas	Alexandre Mota	
51	Projeto de desenvolvimento	Hermano Moura	Jacques Robin
52	Introdução à multimídia	Judith Kelner	
53	Paradigmas de linguagens computacionais	André Santos	Hermano Moura
54	Teoria e implementação de linguagens computacionais	André Santos	
55	Informática teórica	Ruy Barreto de Queiroz	Kátia Guimarães
56	História e futuro da computação	Silvio Meira	
57	Inglês para computação		
58	Cálculo 3A		

Java POI

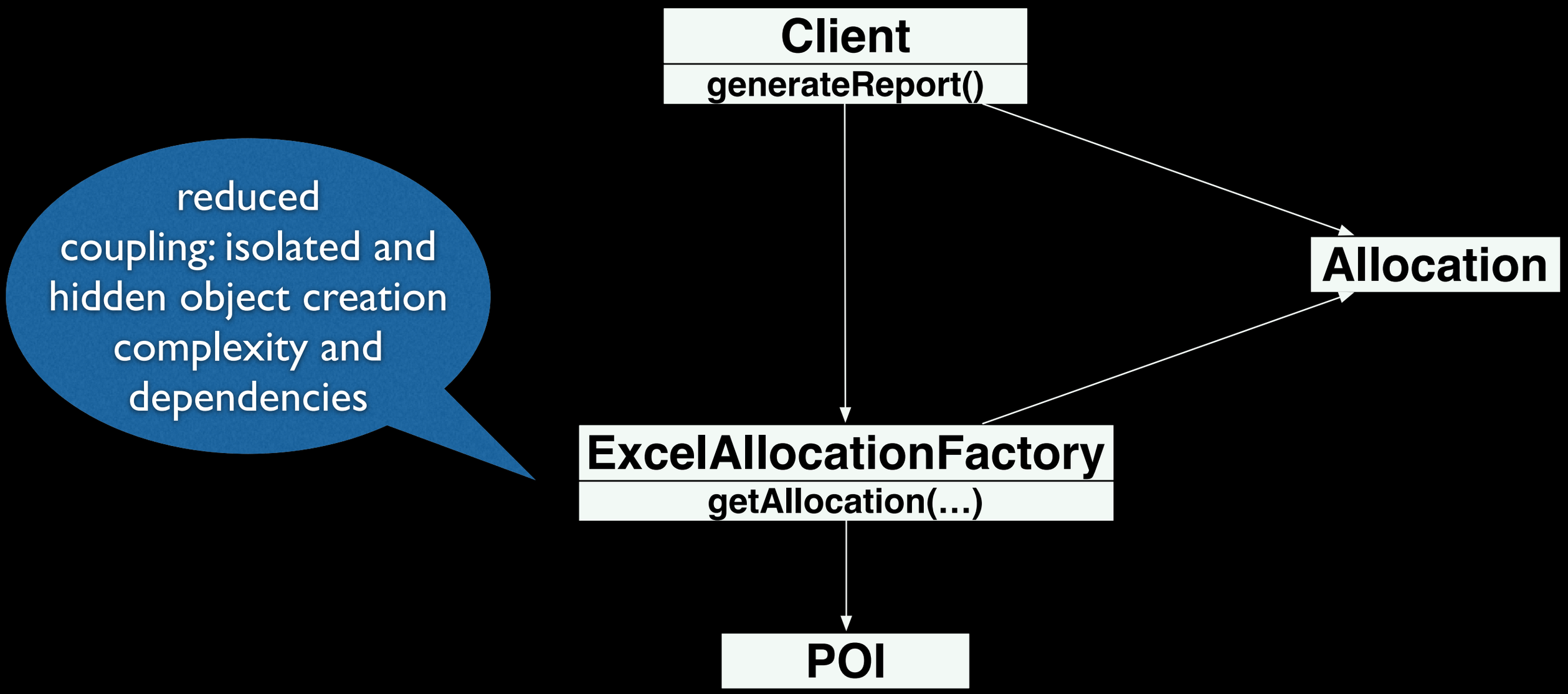
Java APIs Java for manipulating “OLE 2 Compound Document” files, such as xls e doc:

- HSSF (Horrible Spreadsheet Format)
- HDF (Horrible Document Format)
- HPSF (Horrible Property Set Format)

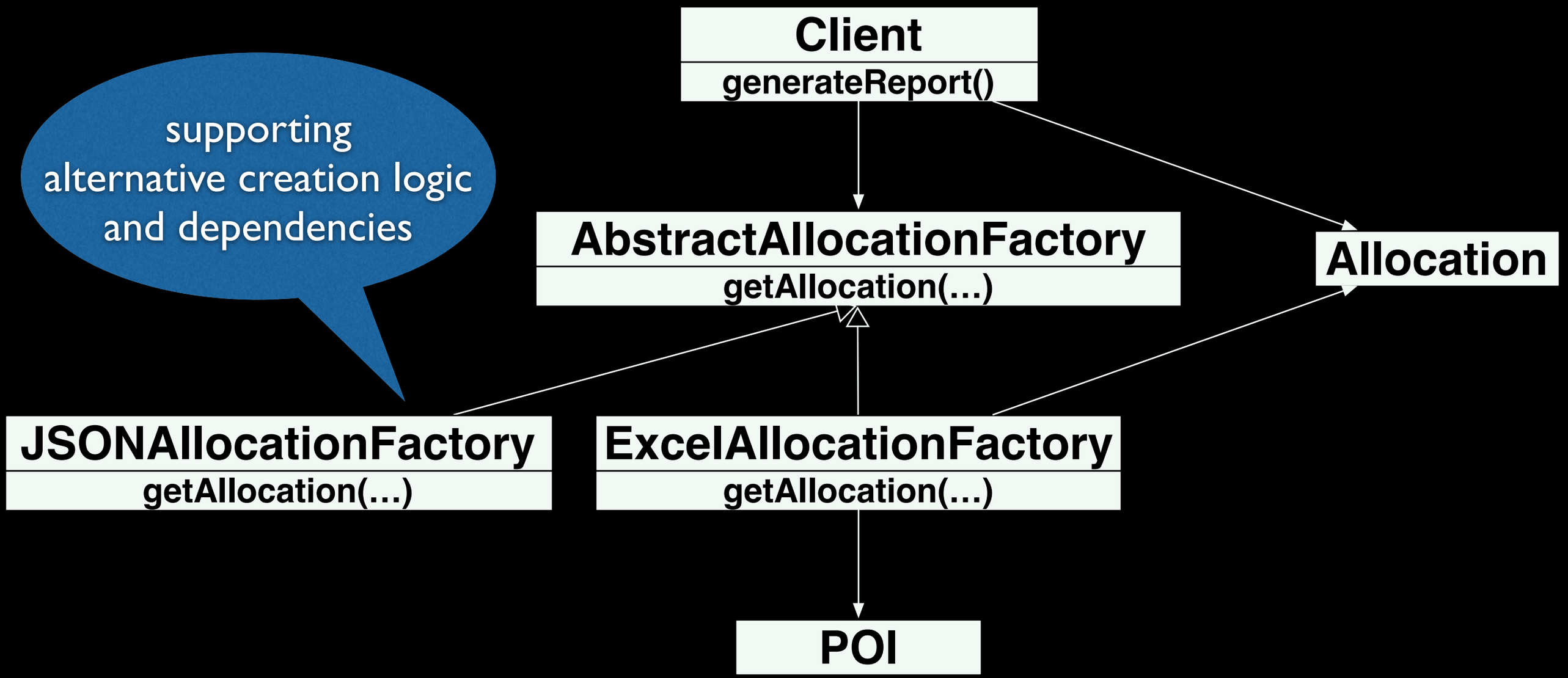
Reduce coupling to library dependencies is often a good idea

Allocation object creation is not simple, and depends on spreadsheet format

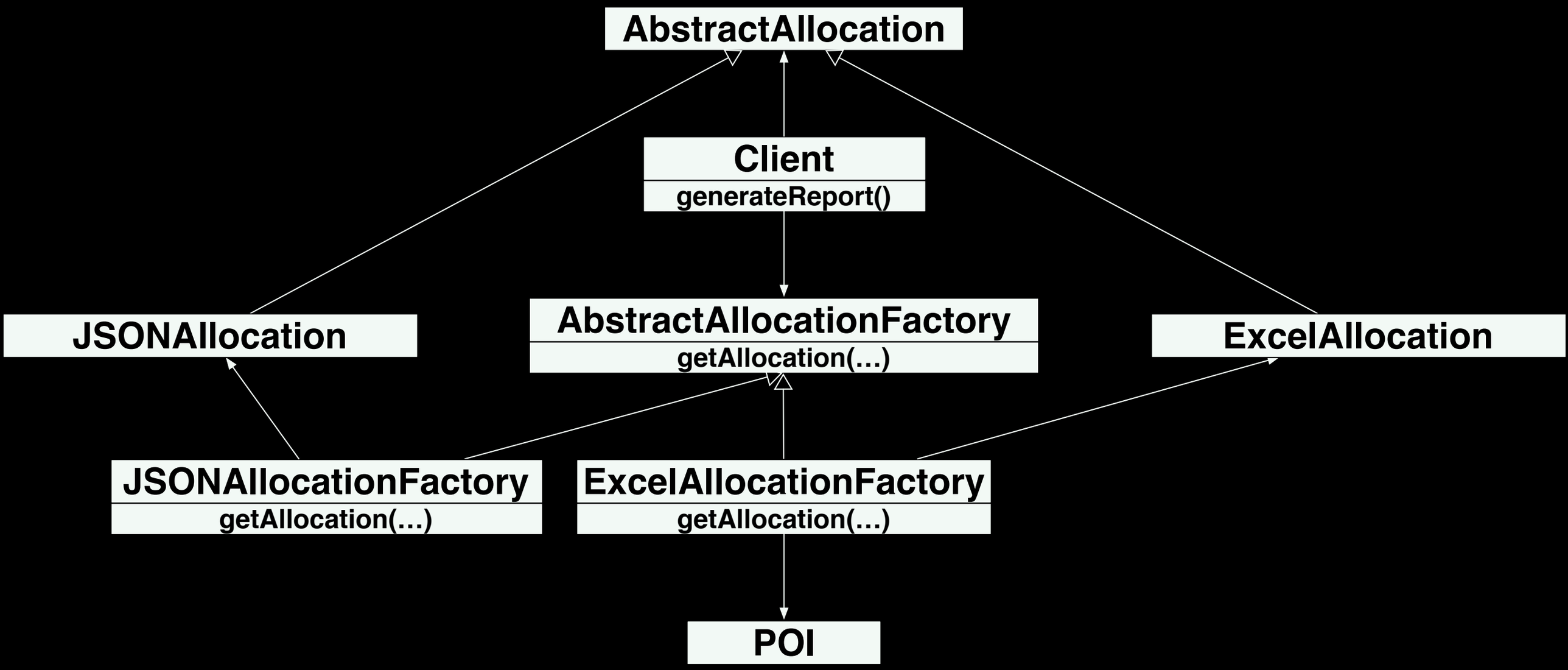
Factory pattern for using POI



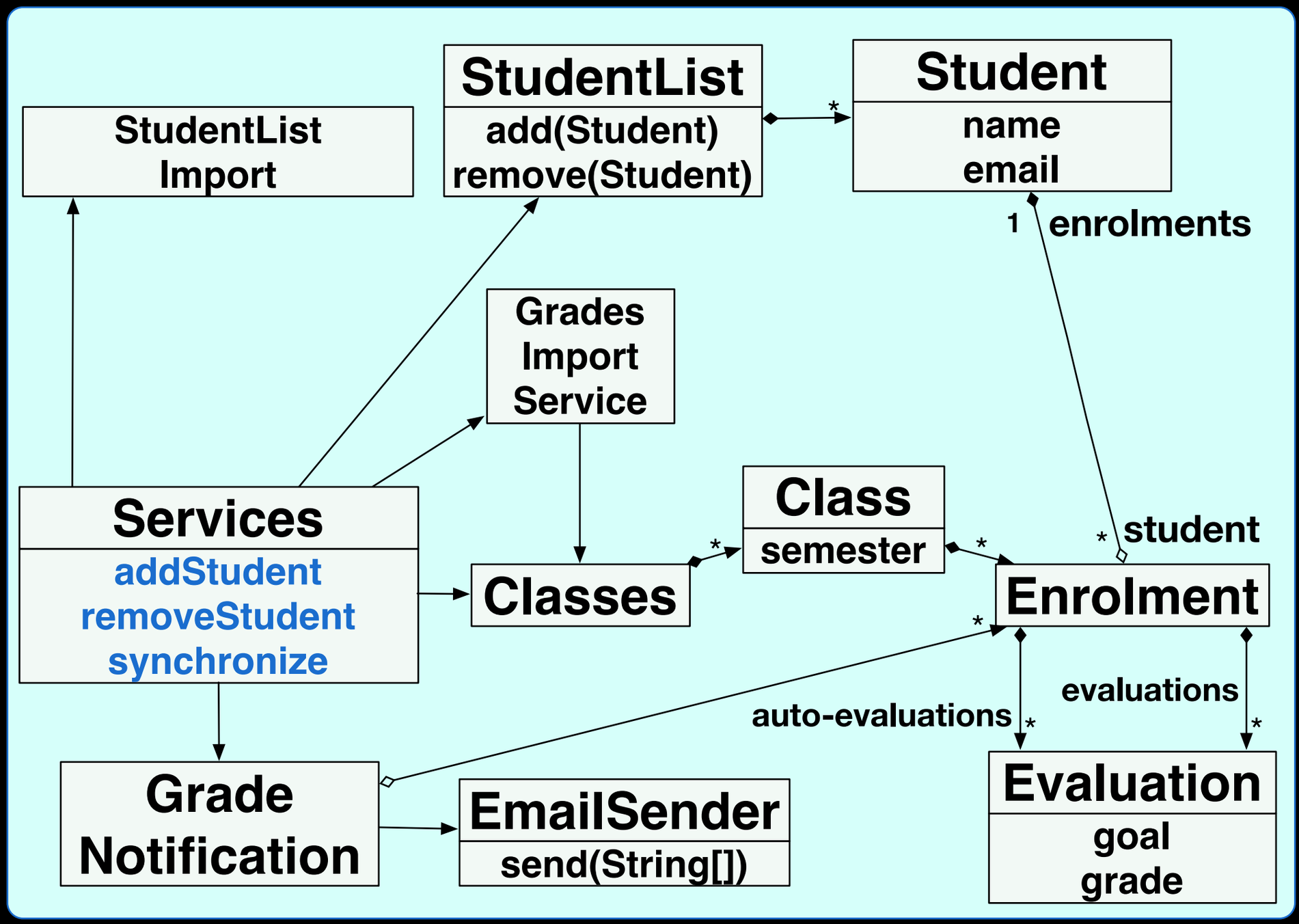
Abstract factory pattern



Further abstracting the created object



A similar design might be adequate for StudentListImport



Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

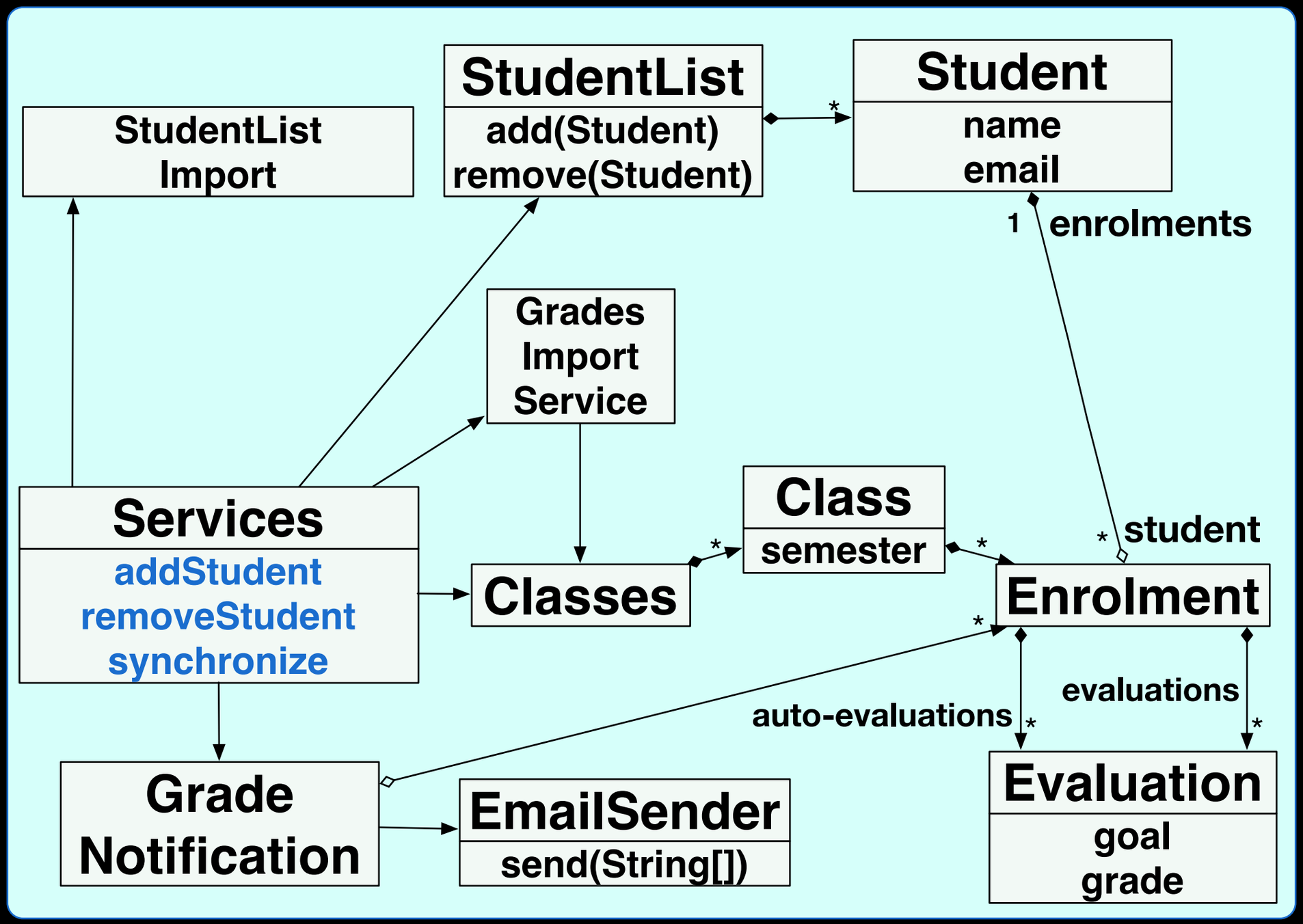
pauloborba.cin.ufpe.br

Decorator

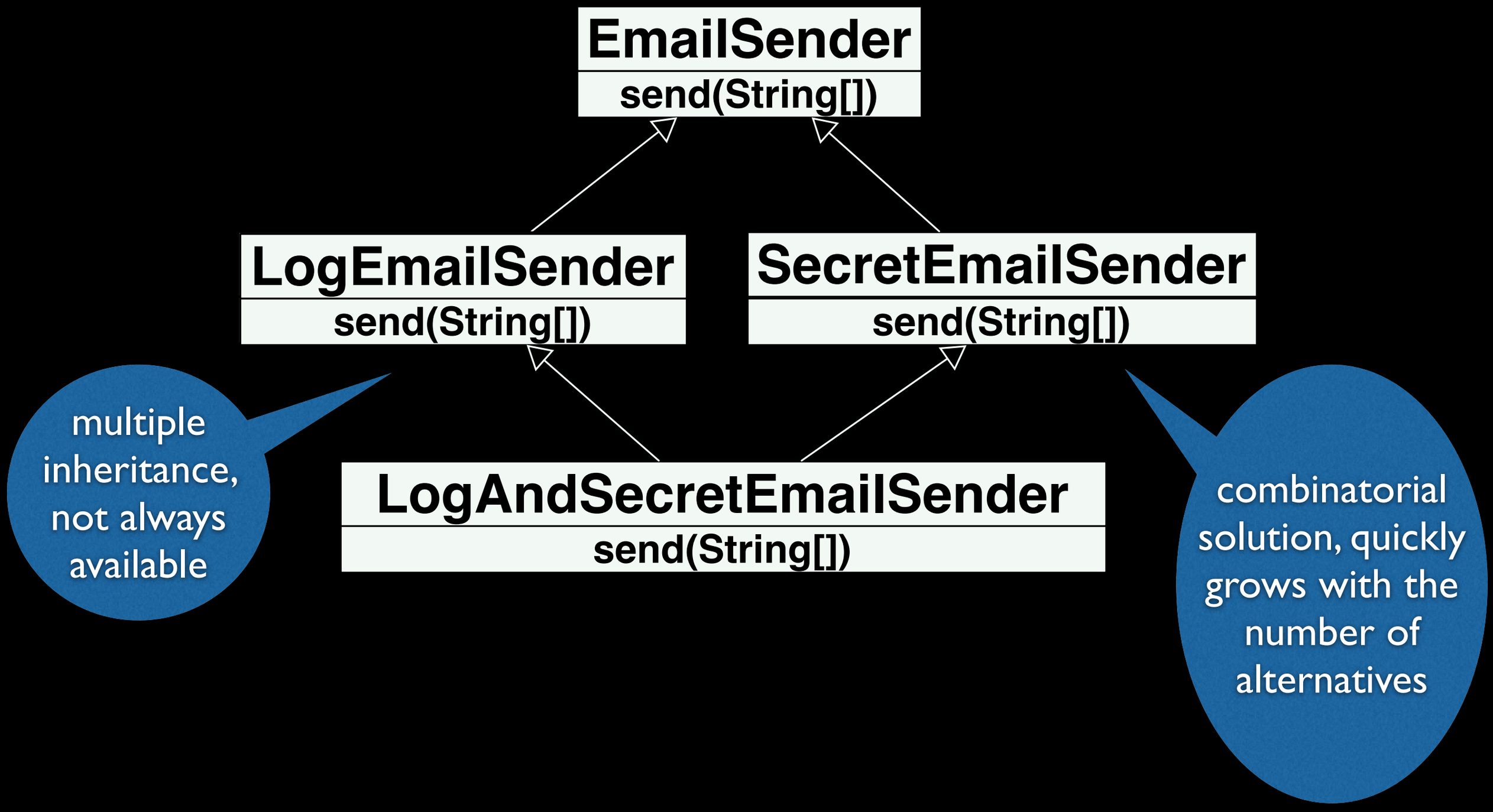
Context

different clients or situations
demand additional behavior,
and combinations

Logging and cyphering emails, but not for all



Problem


EmailSender
send(String[])
LogEmailSender
send(String[])
SecretEmailSender
send(String[])
LogAndSecretEmailSender
send(String[])
multiple inheritance, not always available
combinatorial solution, quickly grows with the number of alternatives

Problem

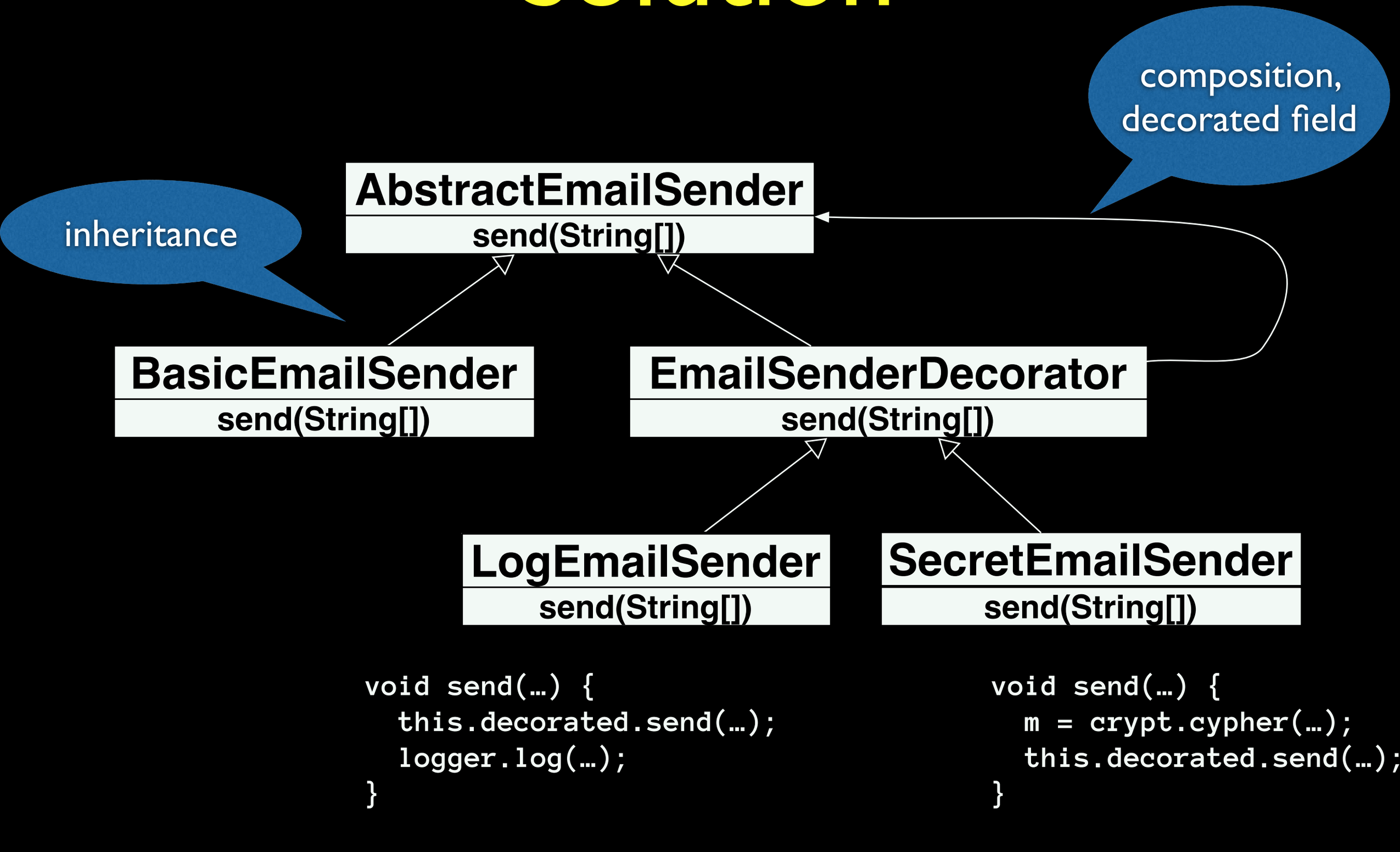
```
void send(…) {
  l = properties.get(“LogEmail”);
  s = properties.get(“SecretEmail”);
  …
  if (l) {
    logger.log(…);
  } if (s) {
    m = crypt.cypher(…);
  } if …
  …
  send email
  …
}
```

lack of cohesion, likely compromises readability

mixes concerns

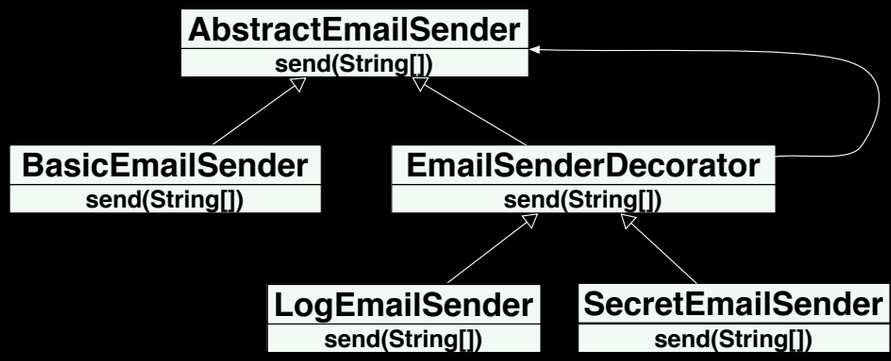
fixed combination order

Solution

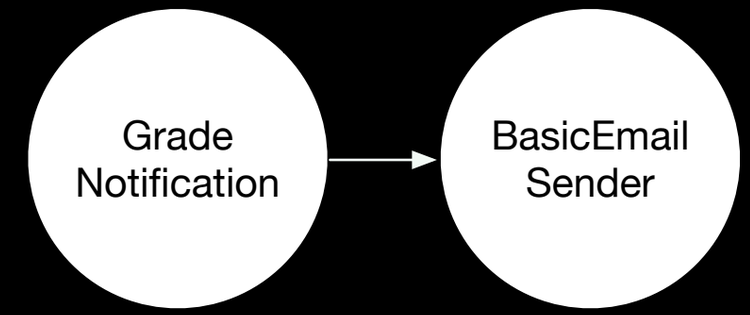


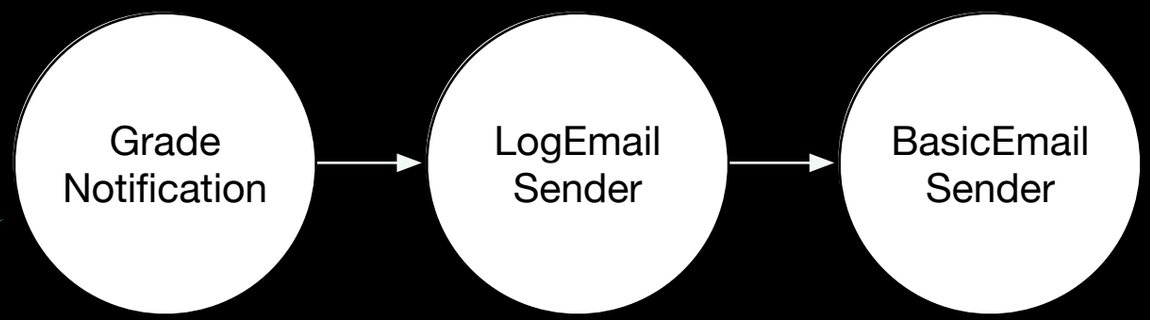

```
void send(…) {
  this.decorated.send(…);
  logger.log(…);
}
```

```
void send(…) {
  m = crypt.cypher(…);
  this.decorated.send(…);
}
```

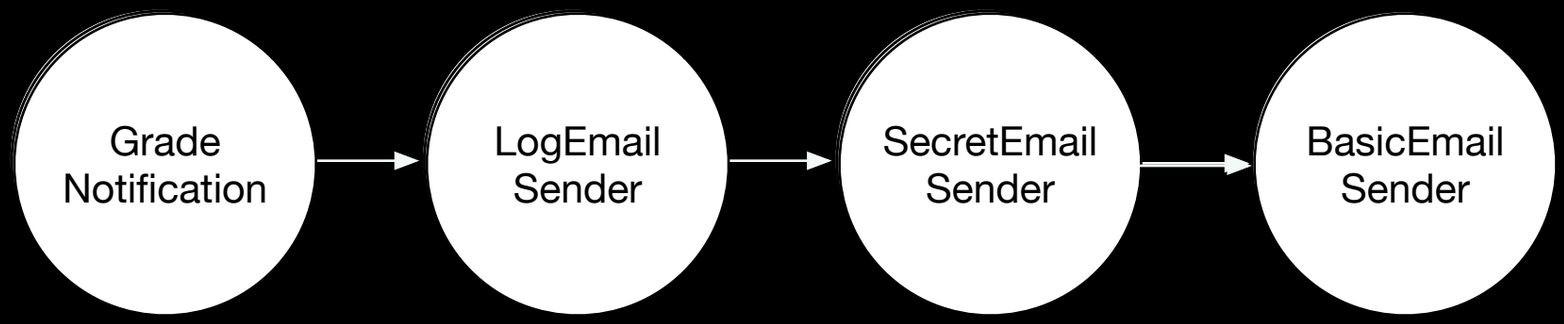


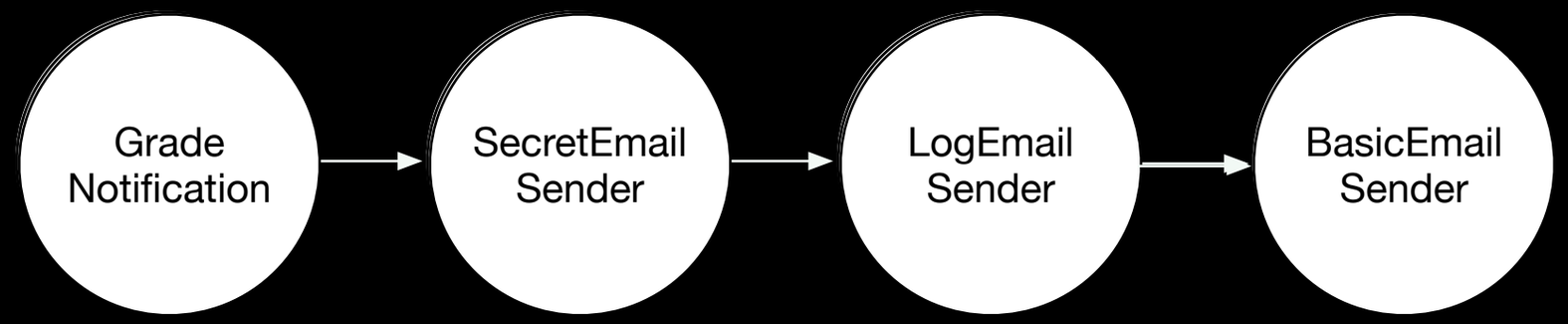
supports multiple combinations with the same class structure





combination can be dynamically changed






```
b = new BasicEmailSender();
l = new LogEmailSender(b);
s = new SecretEmailSender(l);
g = new GradeNotification(s);
```

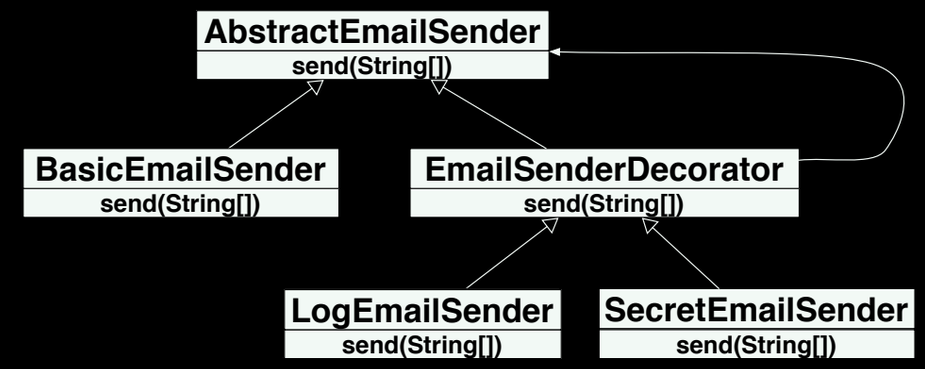
Not often worth to decorate a single (stateless, simple) functionality in a language that supports high order functions

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

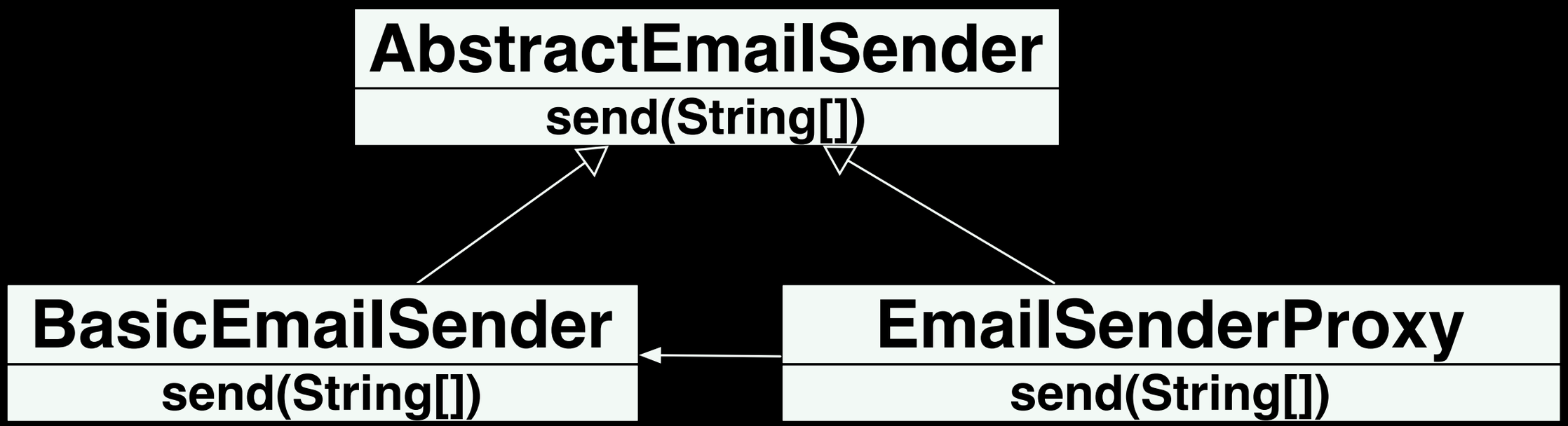
pauloborba.cin.ufpe.br

Chain of responsibility



- Similar structure to decorator
- Different intent
- Handlers versus decoratos
- Handlers can stop the execution chain
- Handlers can execute arbitrary operations (not just additions)

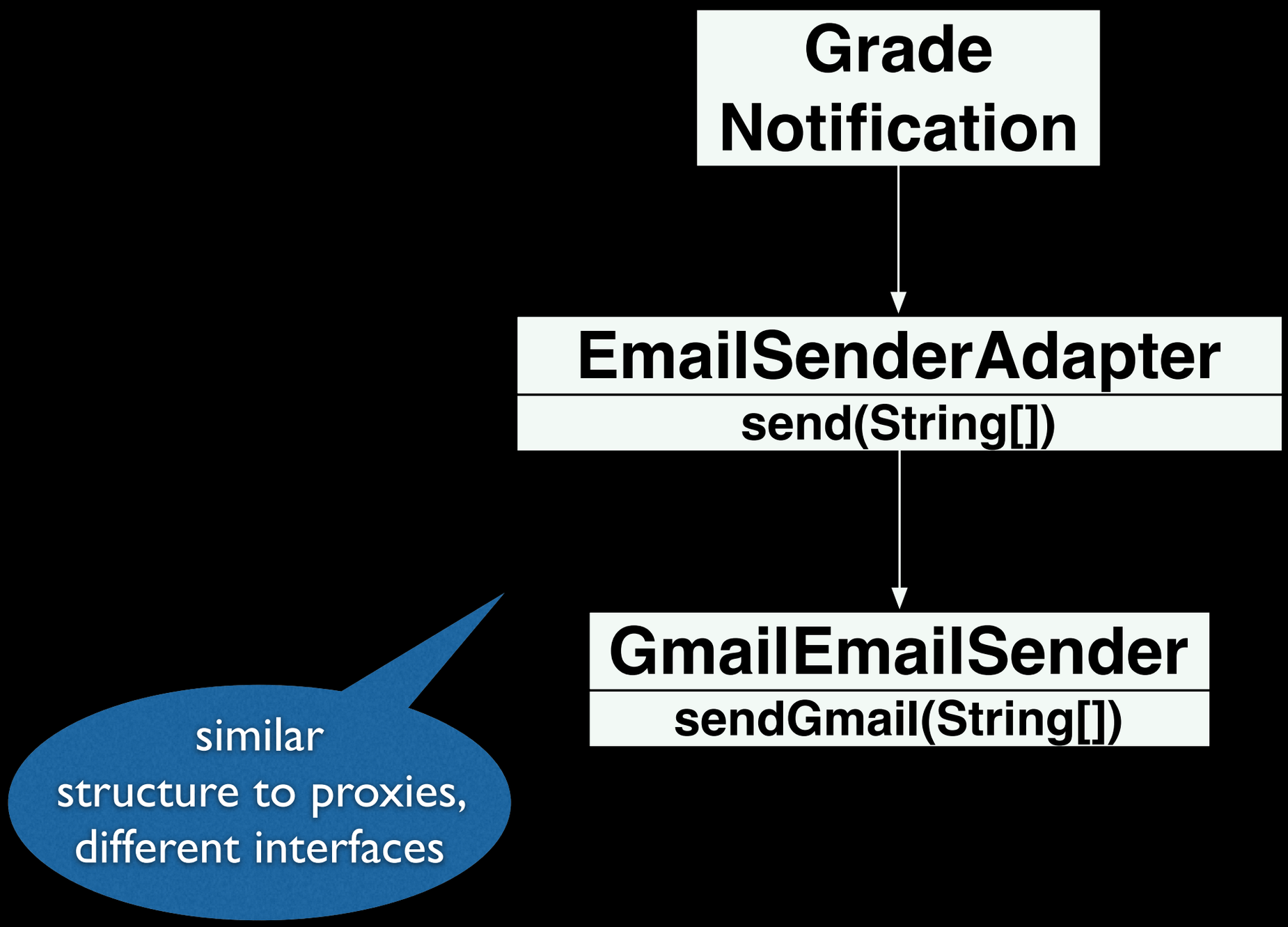
Proxy



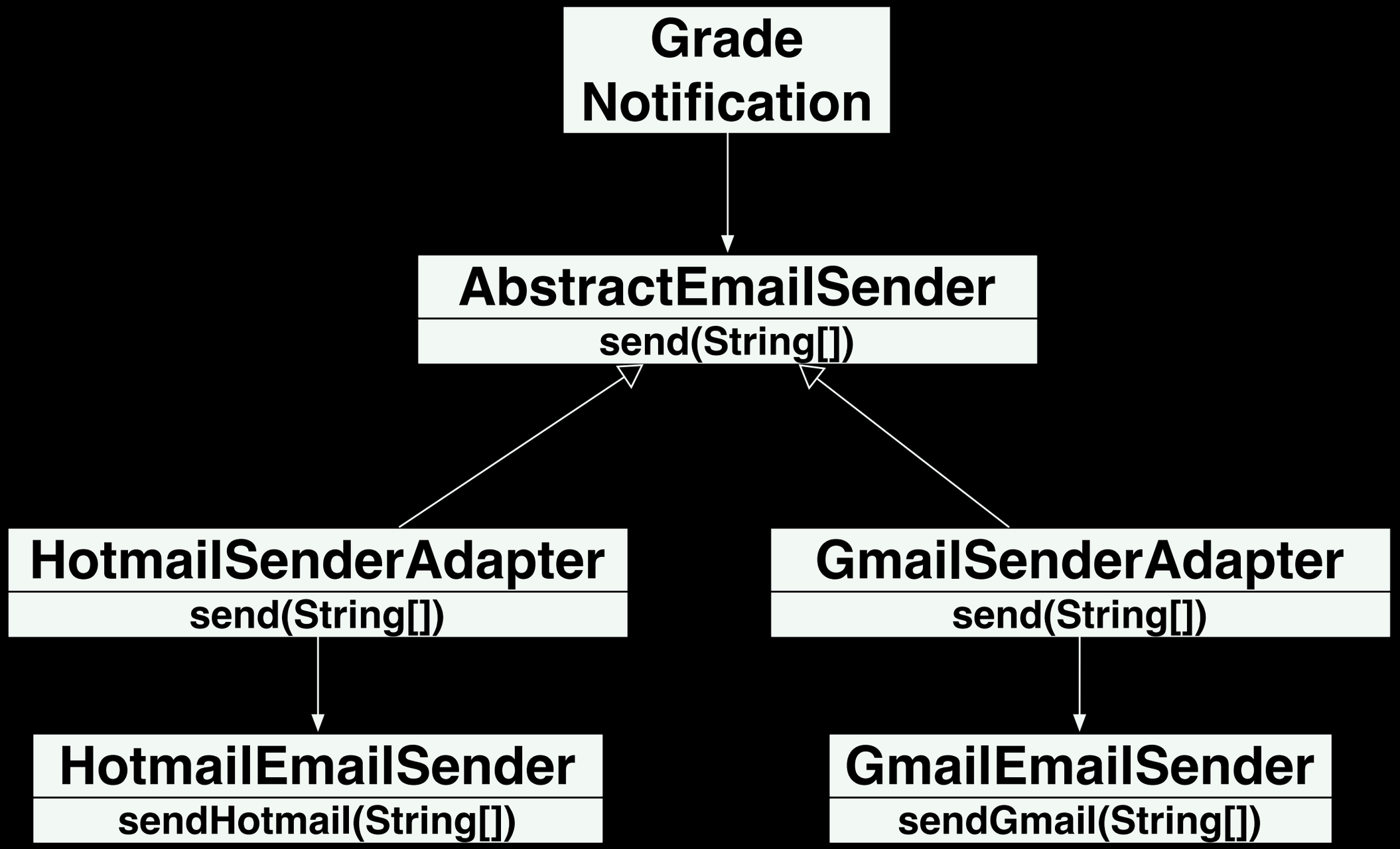
similar structure to decorator

different intent and possibilities; limited combination possibilities

Adapter, wrapper



Client is supposed to conform to a specific interface



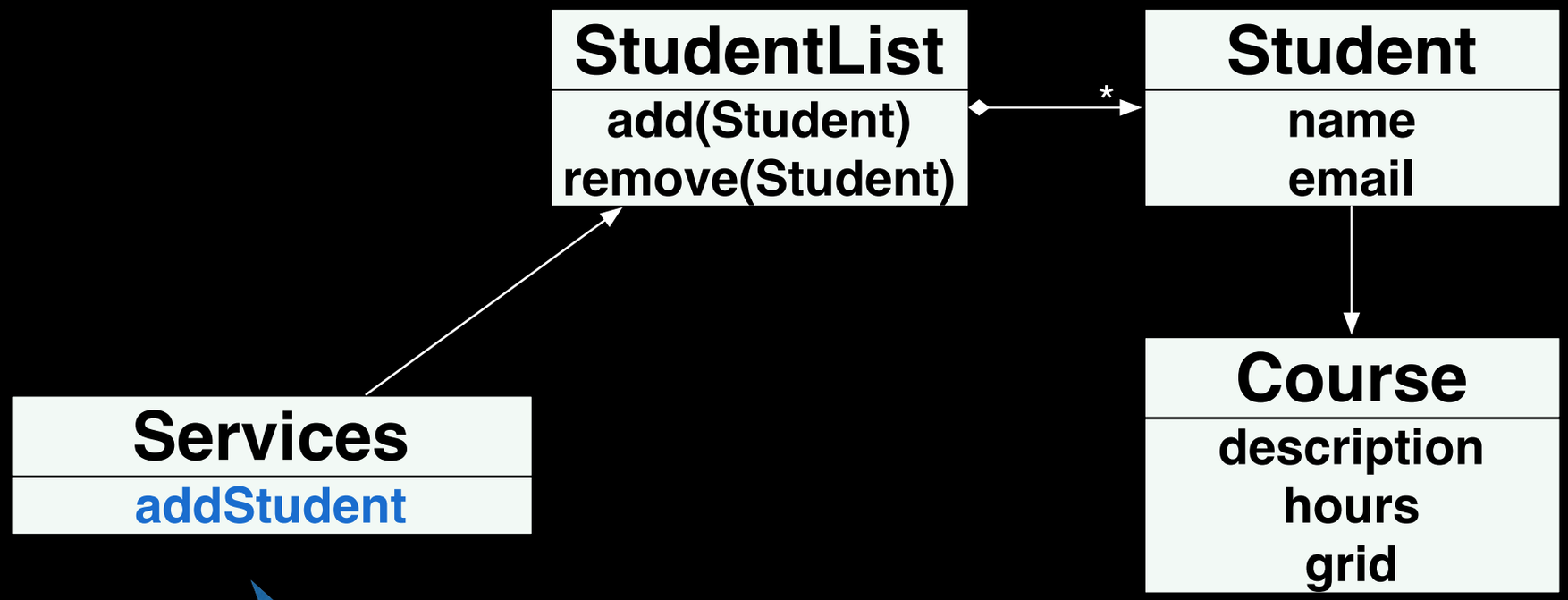
Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

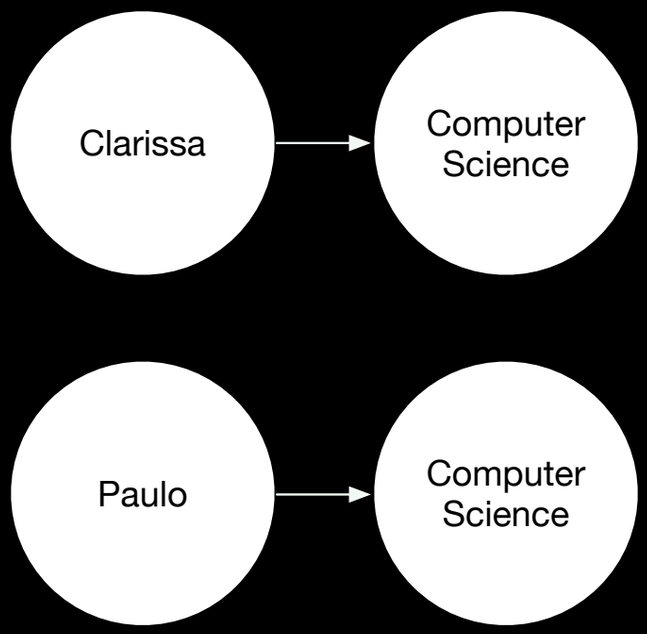
Flyweight

Problem

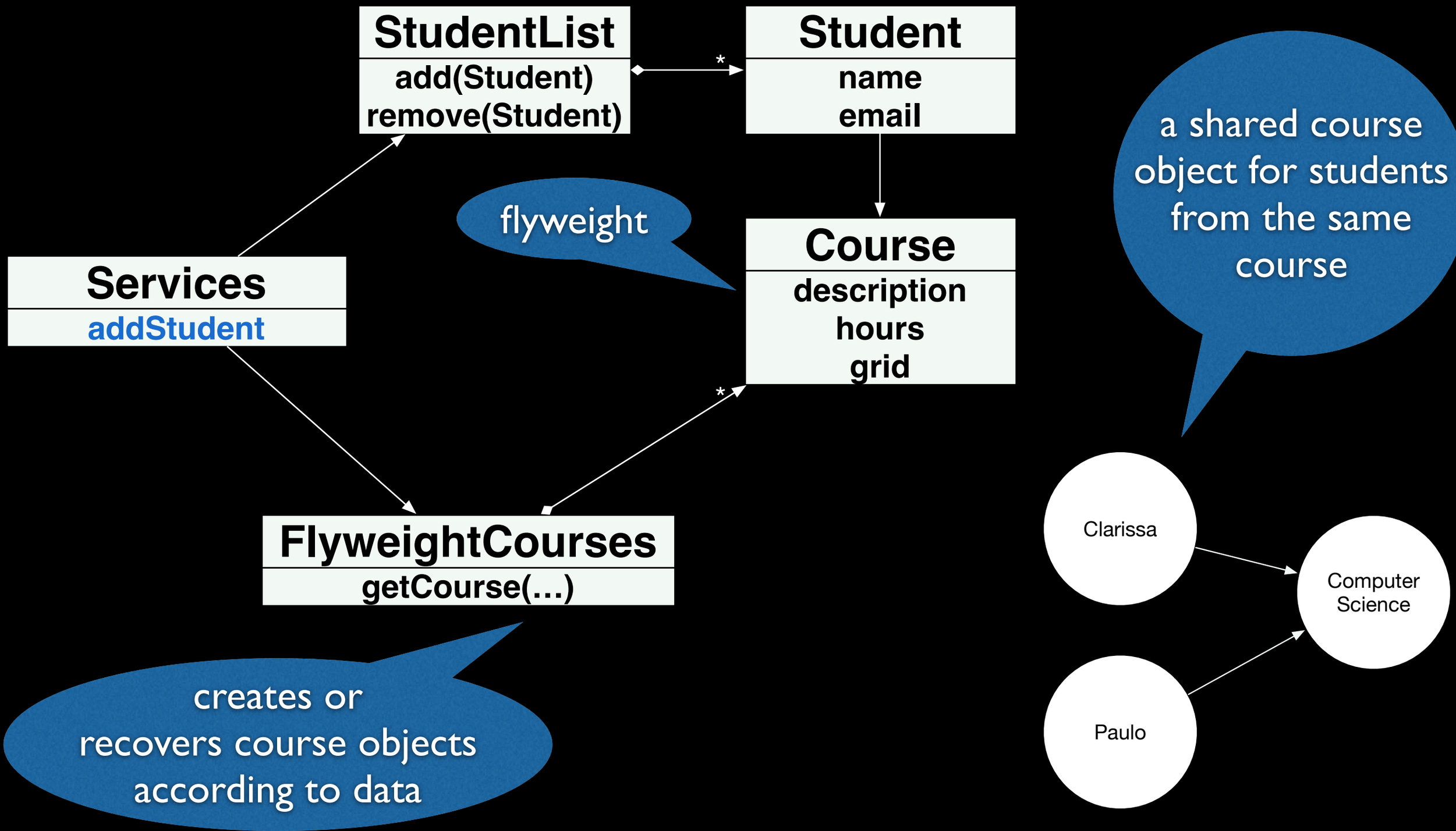


creates student and course objects from provided data

a different, replicated, course object for each student



Solution


StudentList
add(Student)
remove(Student)
*
Student
name
email
flyweight
Course
description
hours
grid
Services
addStudent
*
FlyweightCourses
getCourse(...)
creates or
recovers course objects
according to data
a shared course
object for students
from the same
course
Clarissa
Paulo
Computer
Science

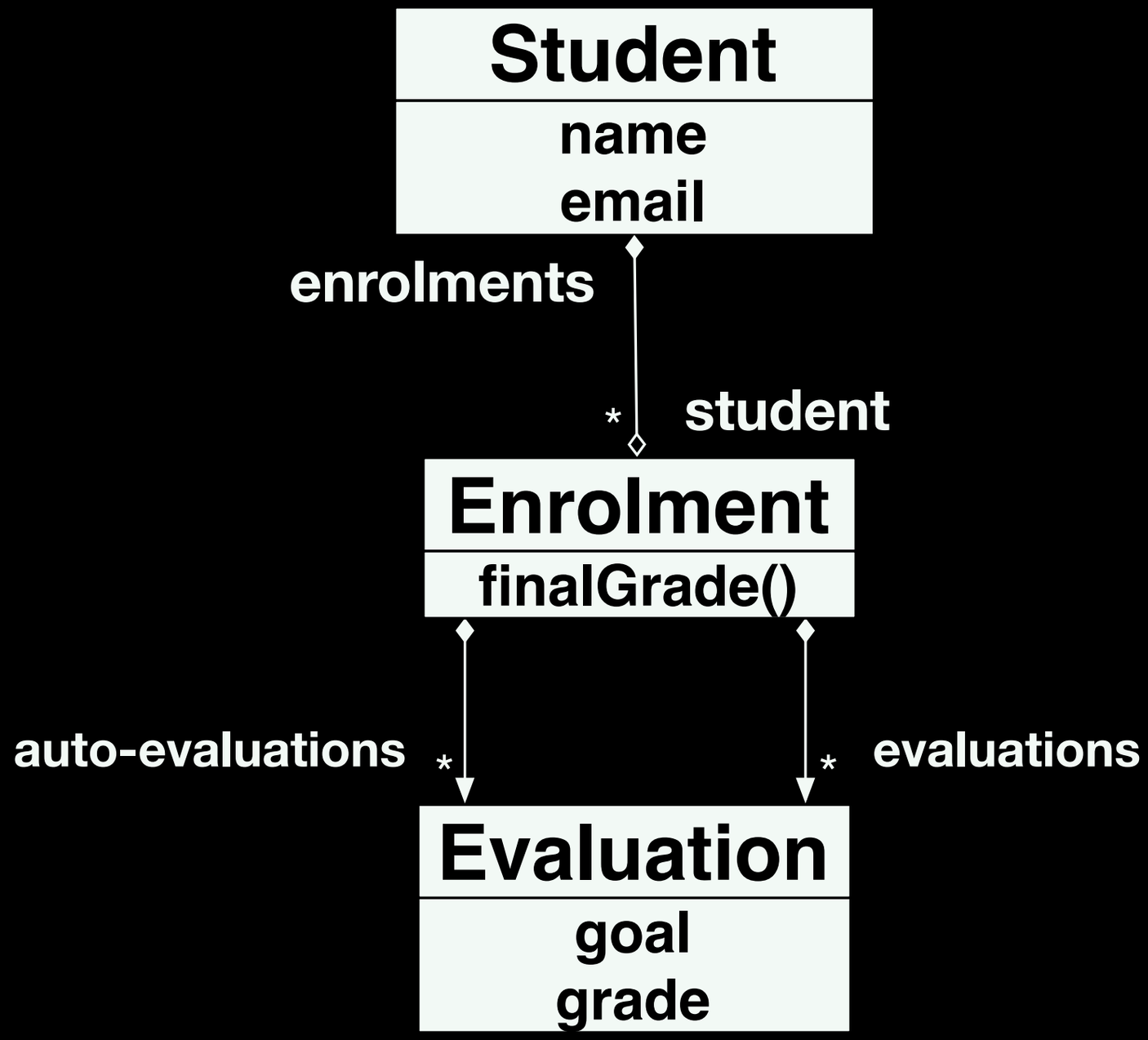
Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Strategy (and command)

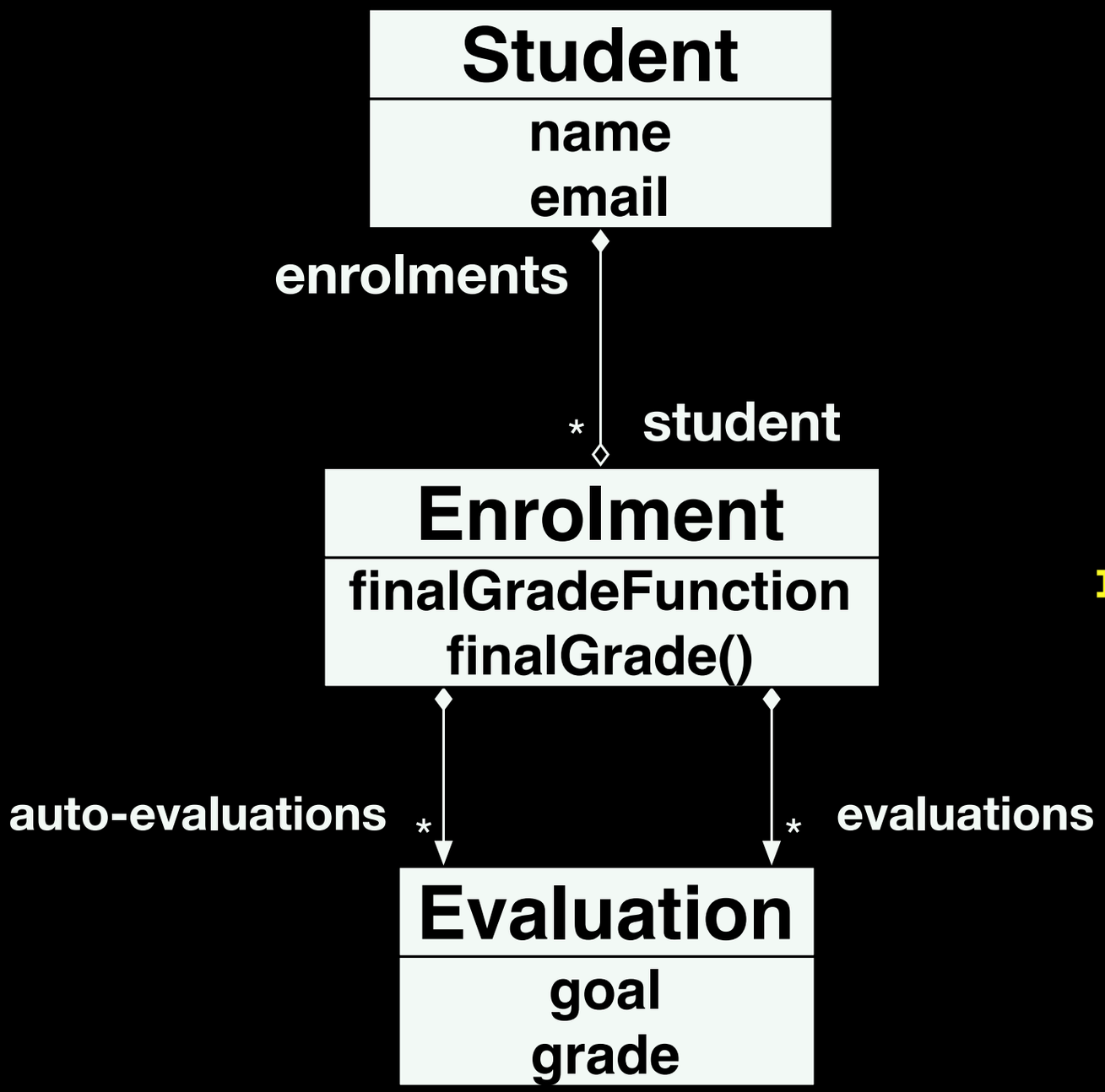
Problem




```
if (…) {…
    finalGrade = …;
} else {…
    finalGrade = …;
}
return finalGrade;
```

fixed or non modular way
for computing grades

Solution

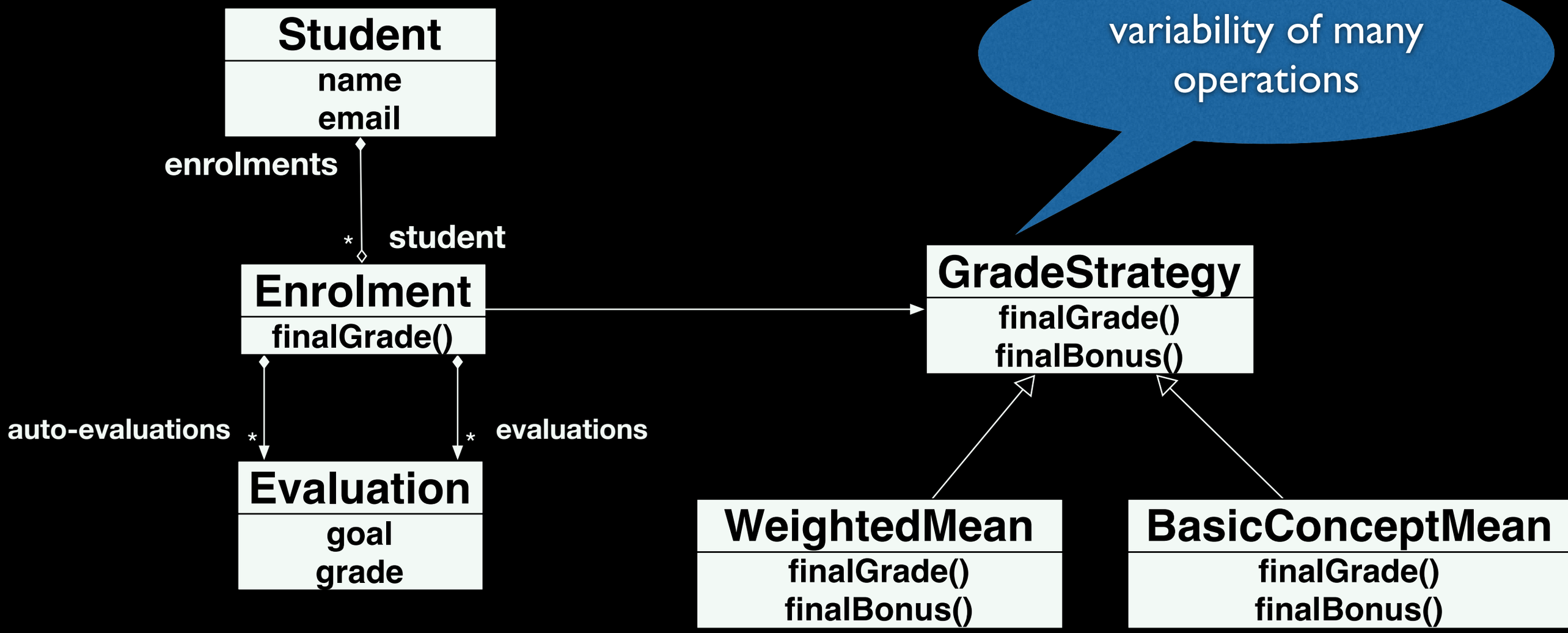


variability of single operation

```
return finalGradeFunction(…);
```

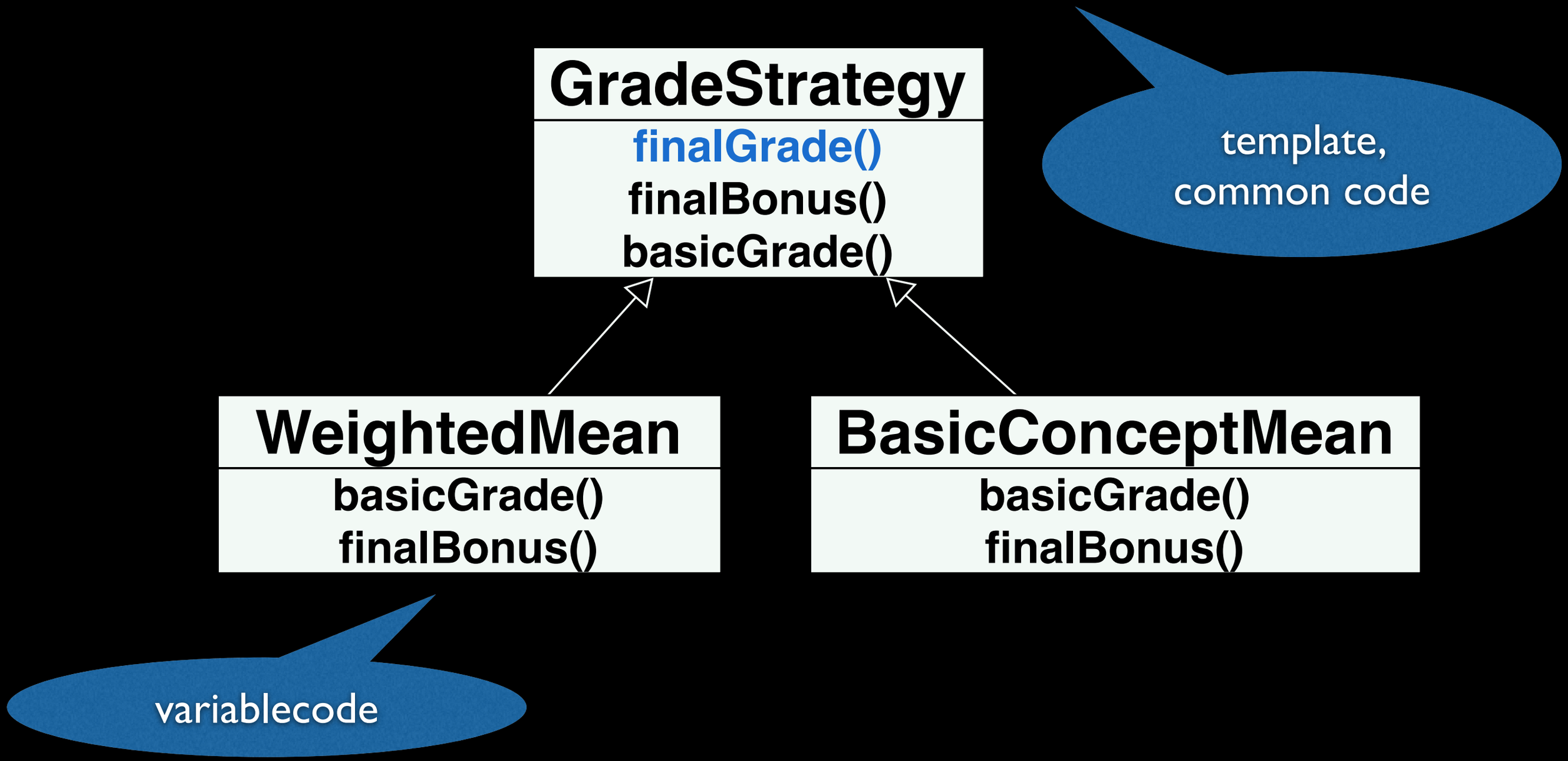
high order languages

Solution


Student
name
email
enrolments
*
student
Enrolment
finalGrade()
auto-evaluations
*
*
evaluations
Evaluation
goal
grade
variability of many operations
GradeStrategy
finalGrade()
finalBonus()
WeightedMean
finalGrade()
finalBonus()
BasicConceptMean
finalGrade()
finalBonus()

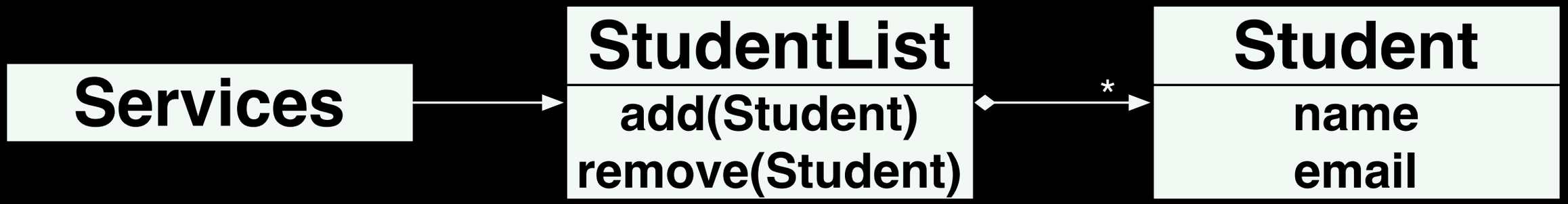
Template method

```
return this.basicGrade() + this.finalBonus();
```

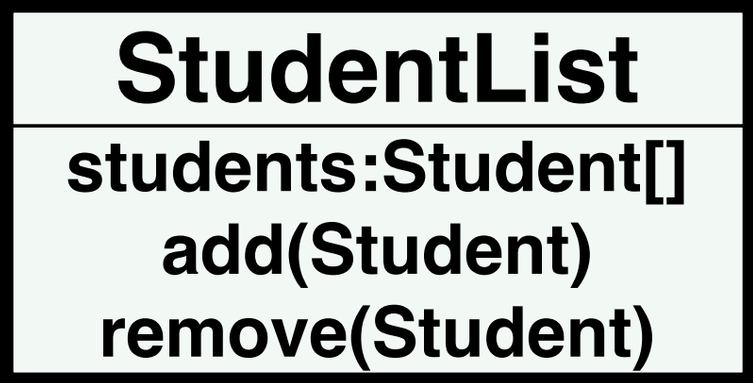


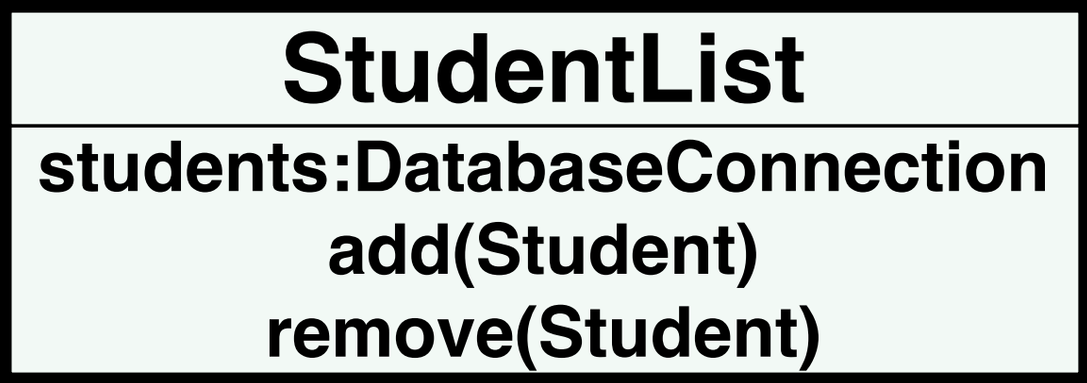
Bridge

Problem

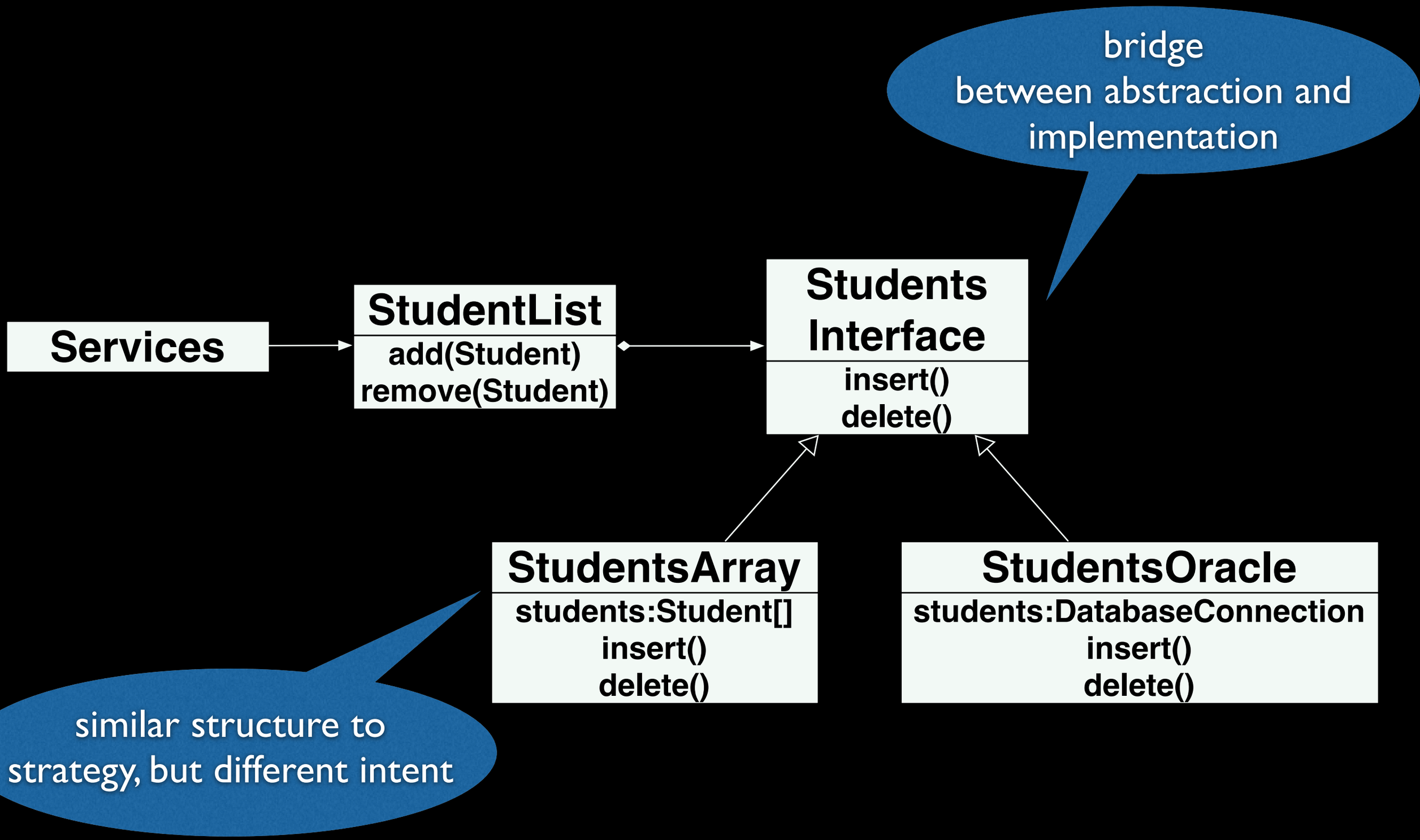


abstraction implemented
in a fixed way

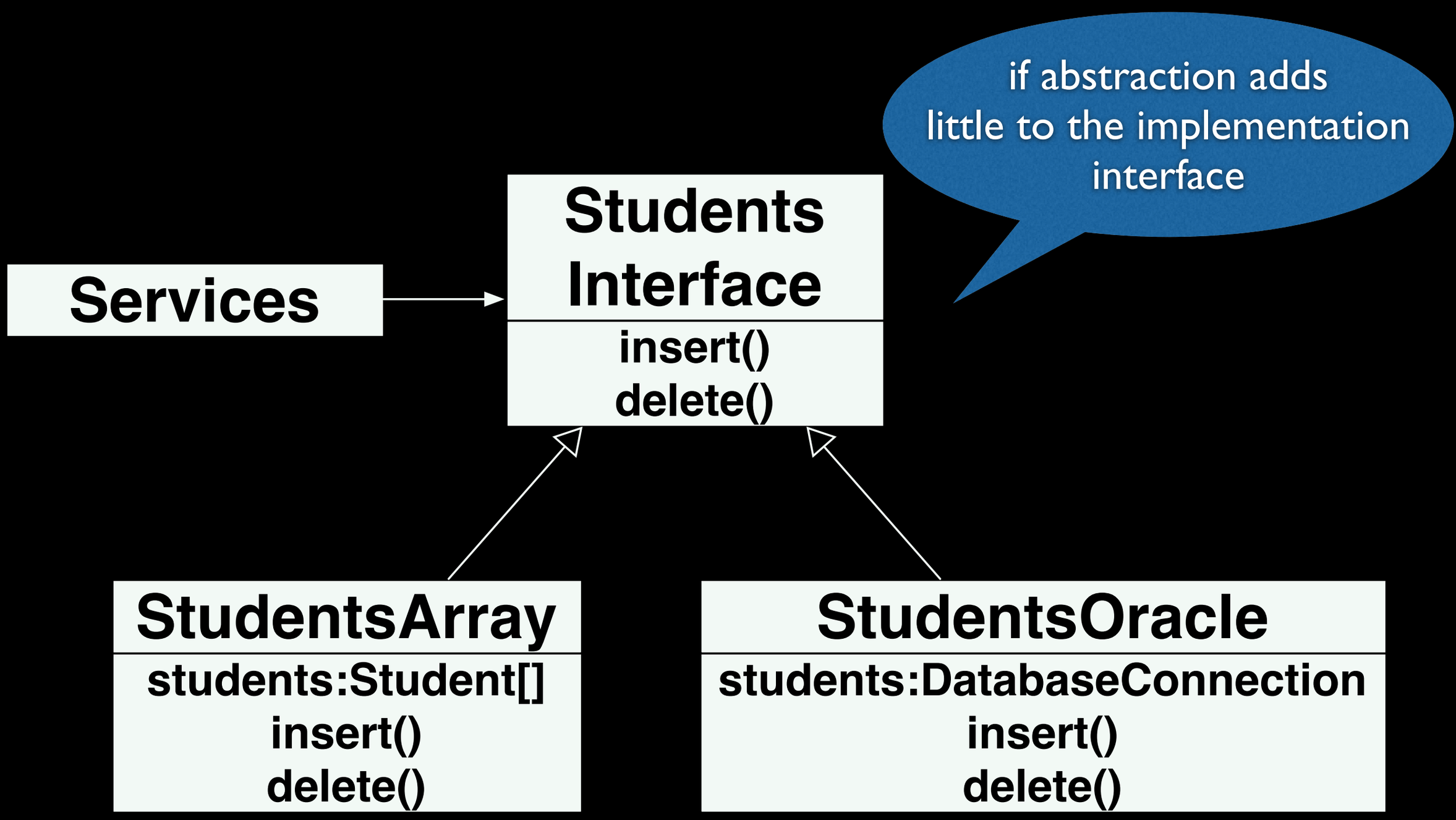




Solution

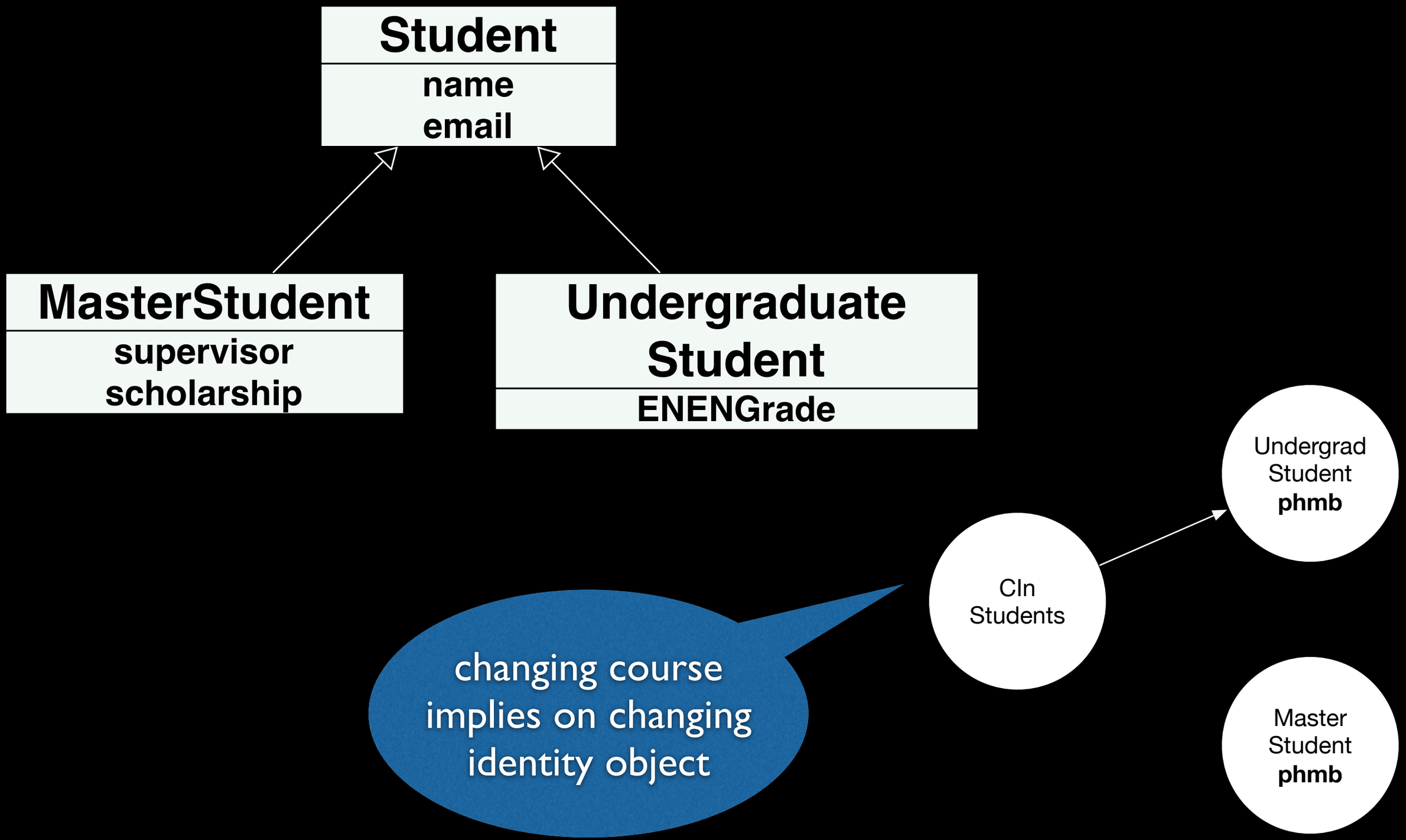

bridge
between abstraction and
implementation
Services
StudentList
add(Student)
remove(Student)
Students
Interface
insert()
delete()
StudentsArray
students:Student[]
insert()
delete()
StudentsOracle
students:DatabaseConnection
insert()
delete()
similar structure to
strategy, but different intent

Solution

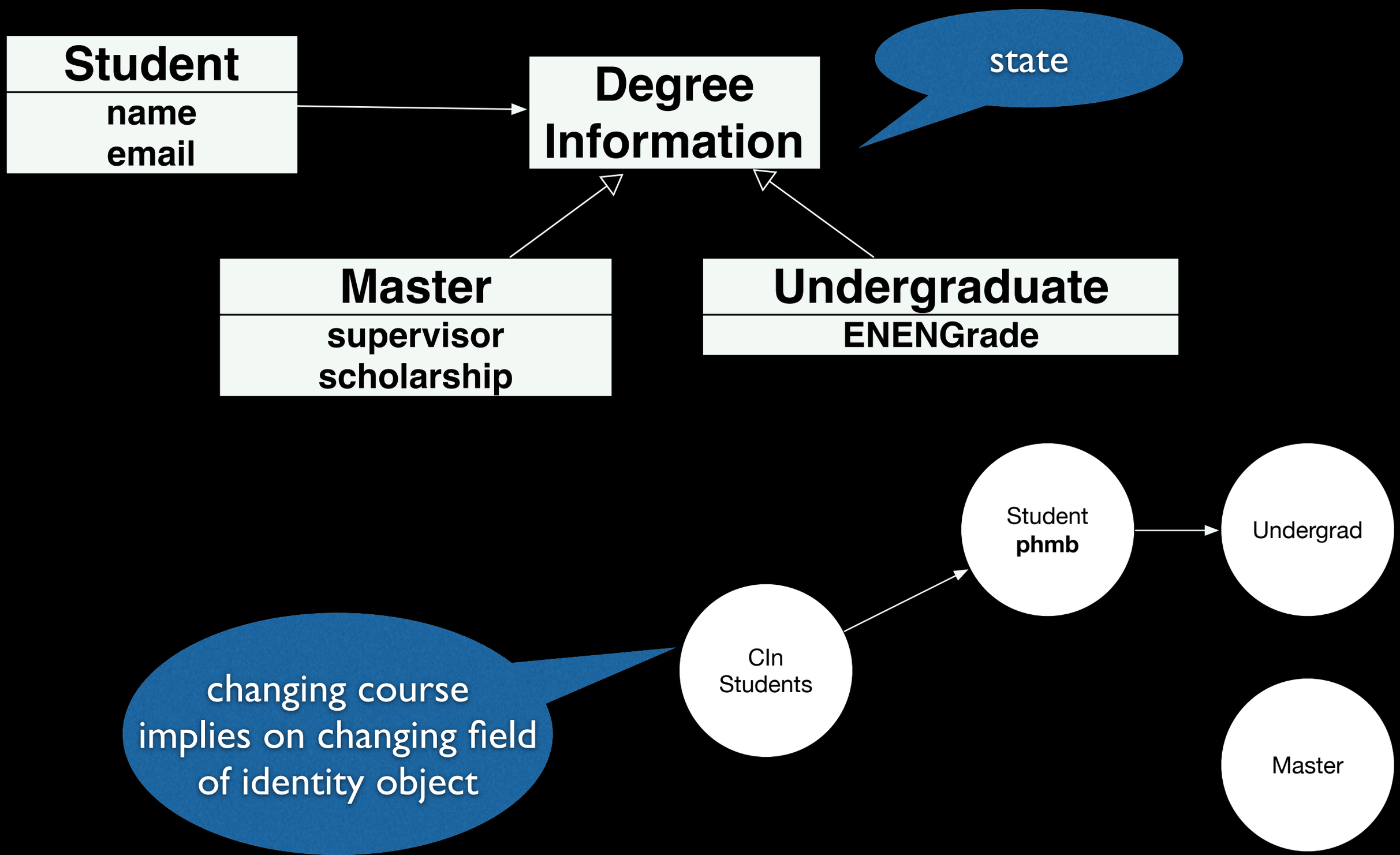

if abstraction adds
little to the implementation
interface
Services
Students
Interface
insert()
delete()
StudentsArray
students:Student[]
insert()
delete()
StudentsOracle
students:DatabaseConnection
insert()
delete()

State

Problem

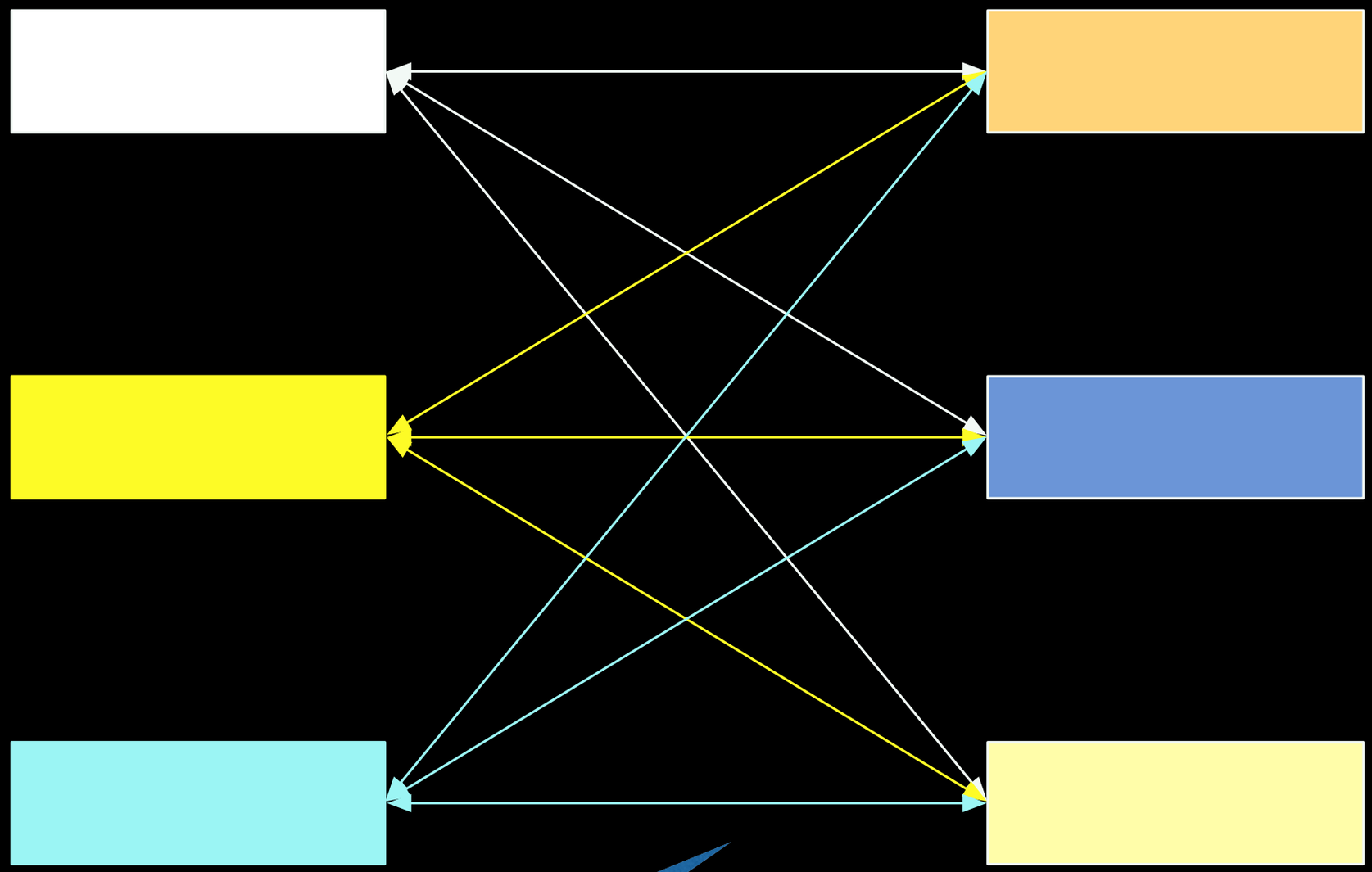

Student
name
email
MasterStudent
supervisor
scholarship
Undergraduate Student
ENENGrade
changing course implies on changing identity object
CIn Students
Undergrad Student
phmb
Master Student
phmb

Solution


Student
name
email
Degree Information
state
Master
supervisor
scholarship
Undergraduate
ENENGrade
changing course implies on changing field of identity object
CIn Students
Student
phmb
Undergrad
Master

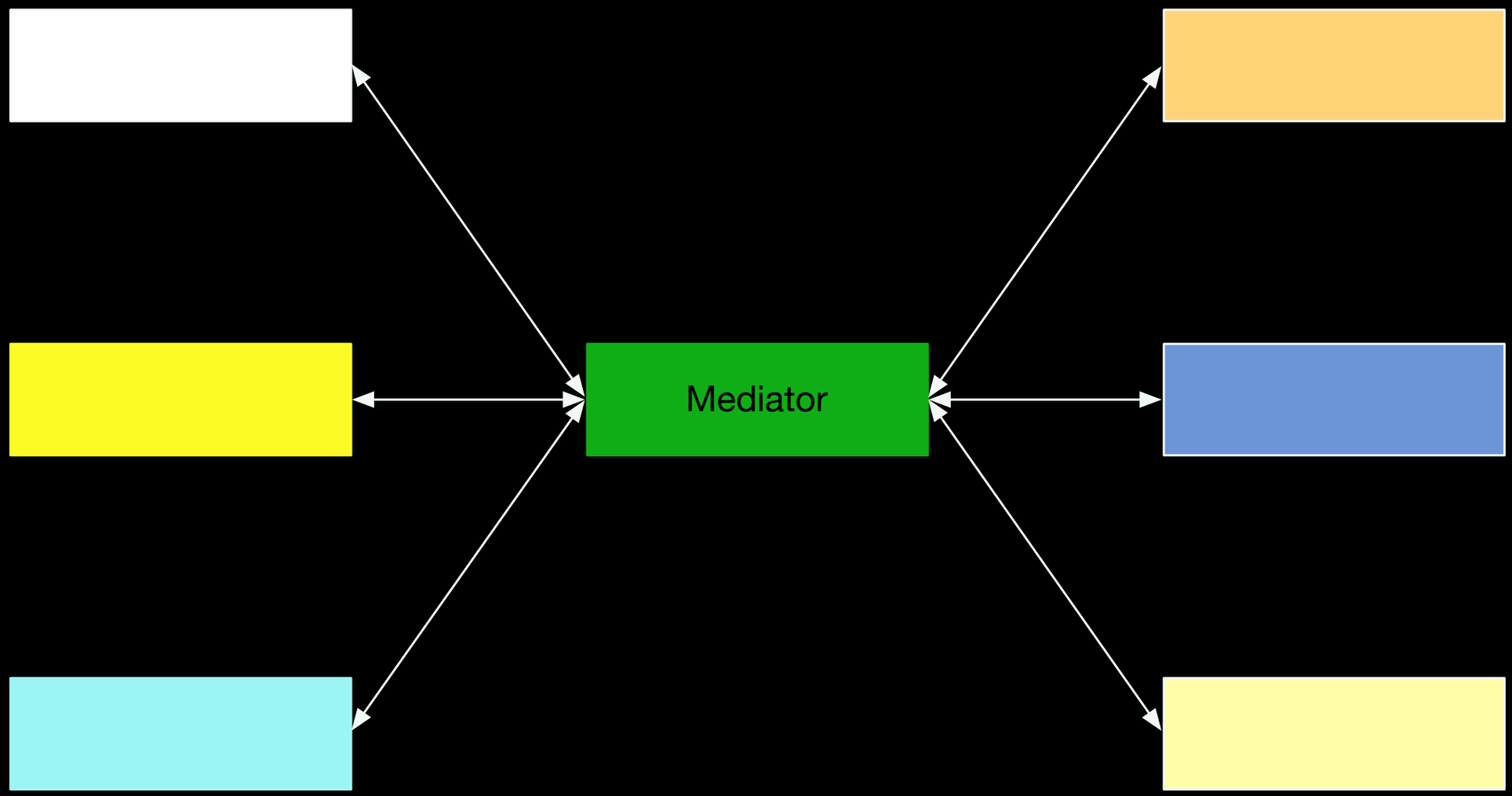
Mediator

Problem

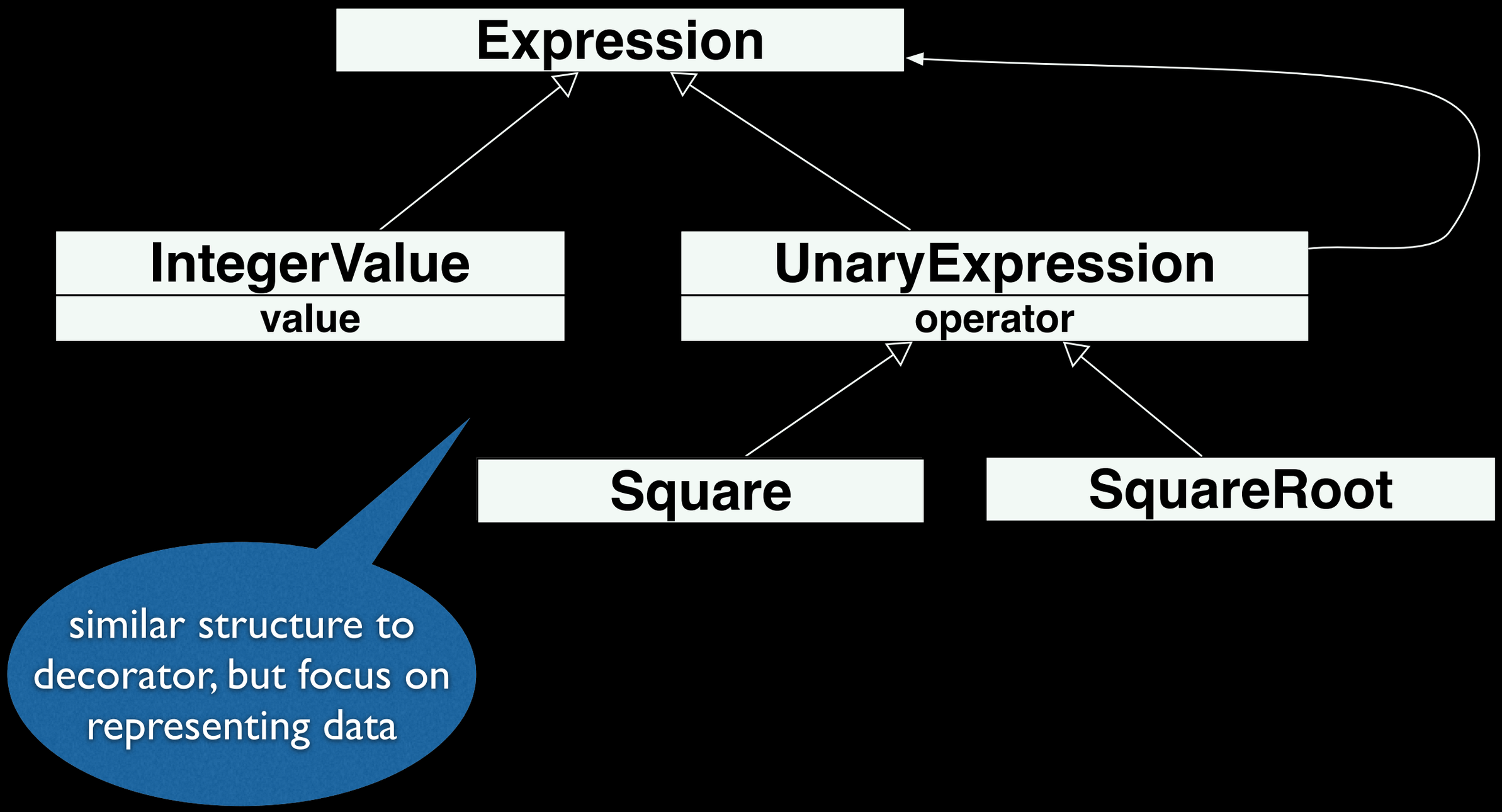


bidirectional dependencies,
contrasting with facade

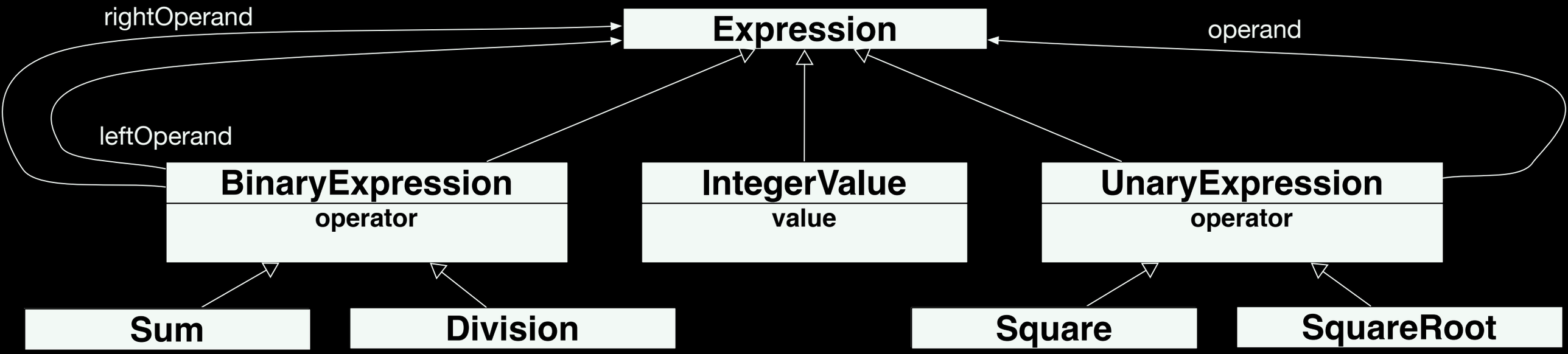
Solution

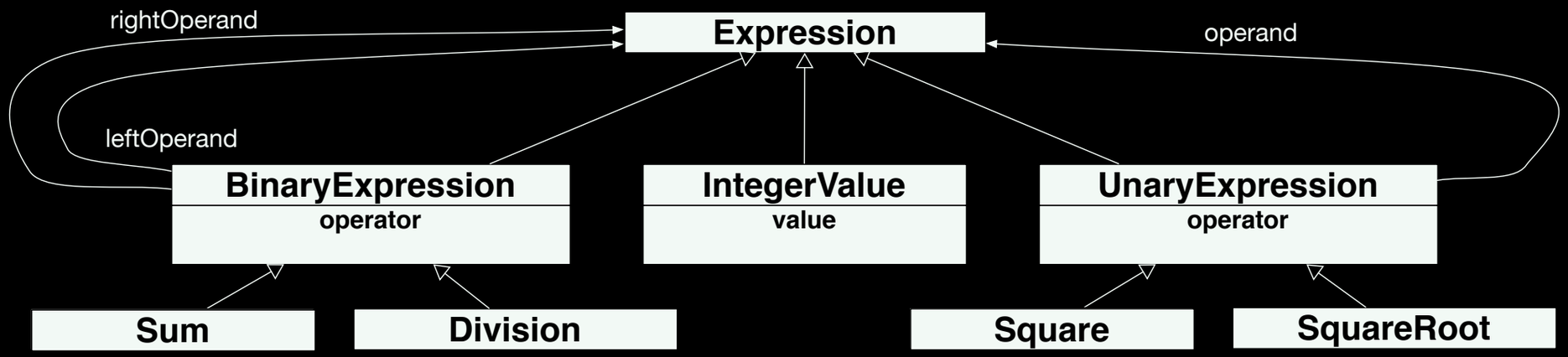


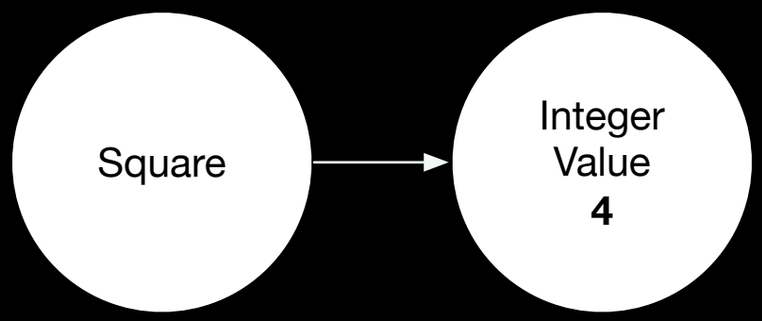
Composite

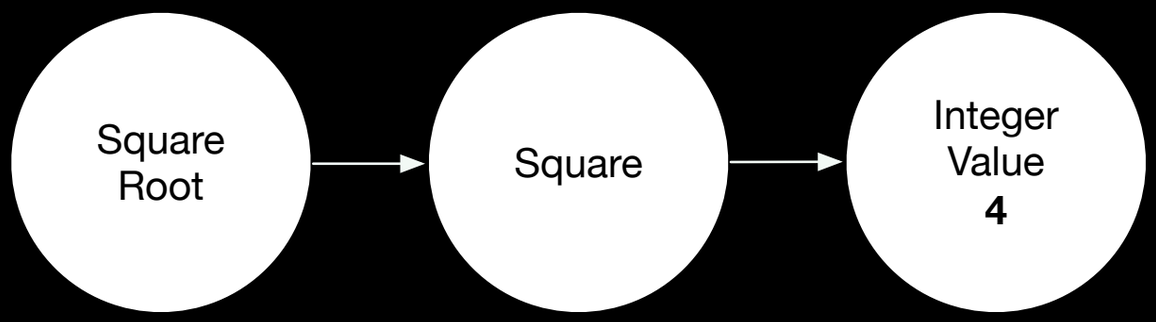

Expression
IntegerValue
value
UnaryExpression
operator
Square
SquareRoot
similar structure to decorator, but focus on representing data

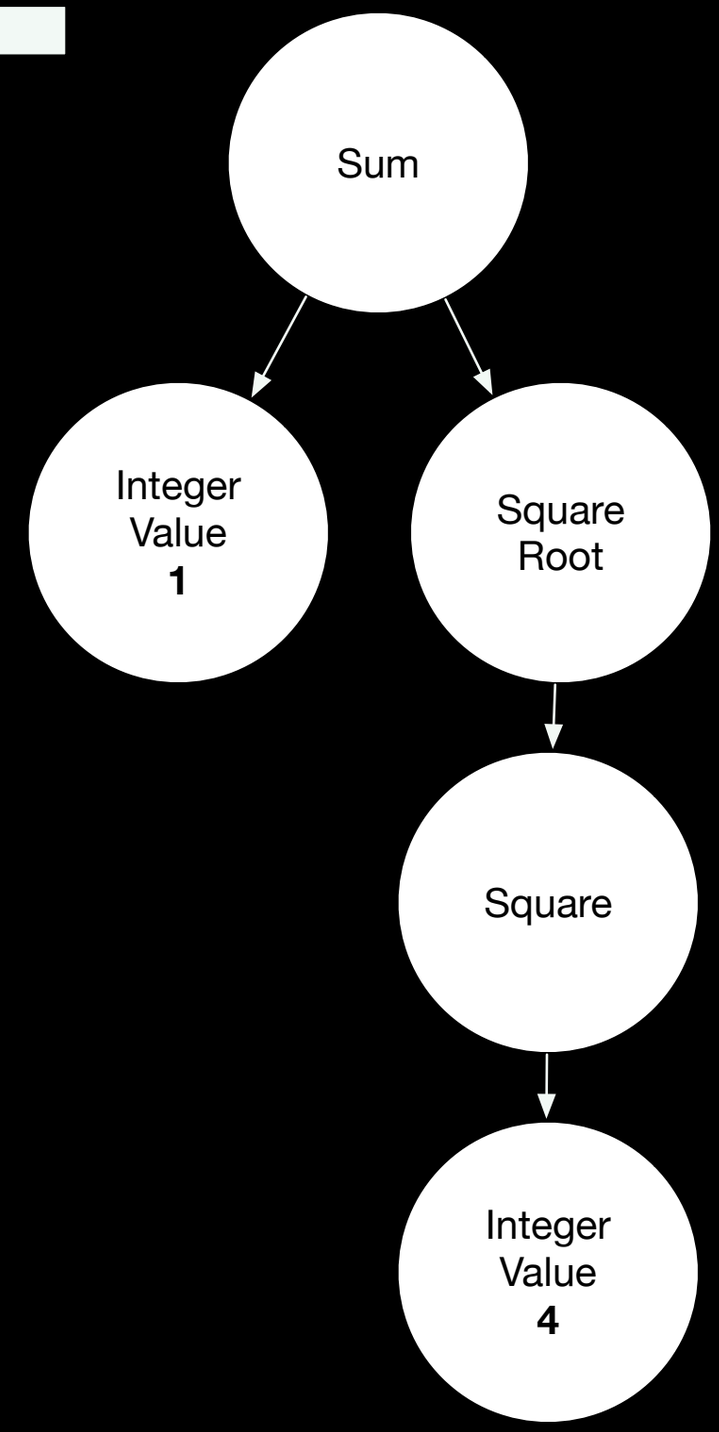
Often more complex structures



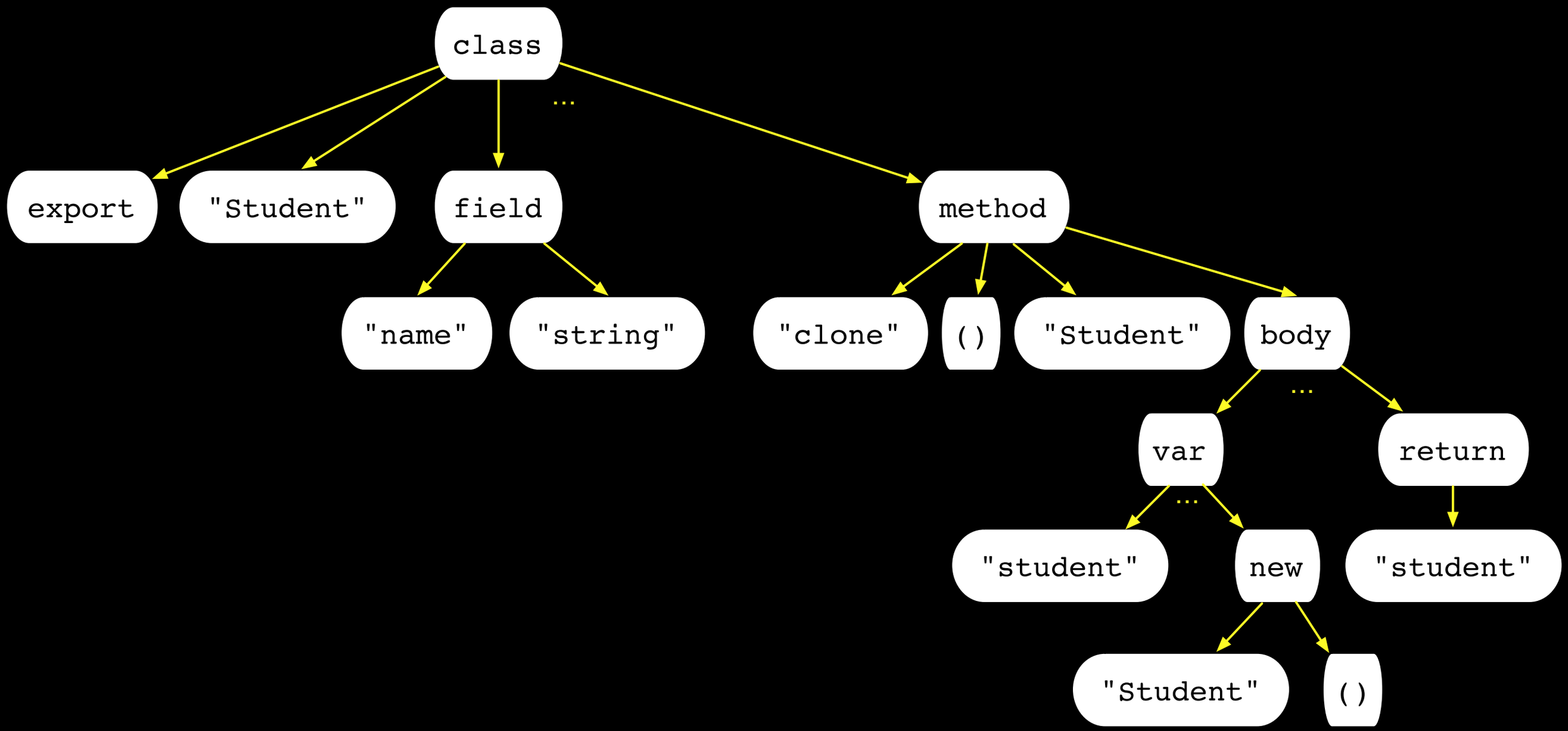

rightOperand
Expression
operand
leftOperand
BinaryExpression
operator
IntegerValue
value
UnaryExpression
operator
Sum
Division
Square
SquareRoot


Square
Integer
Value
4

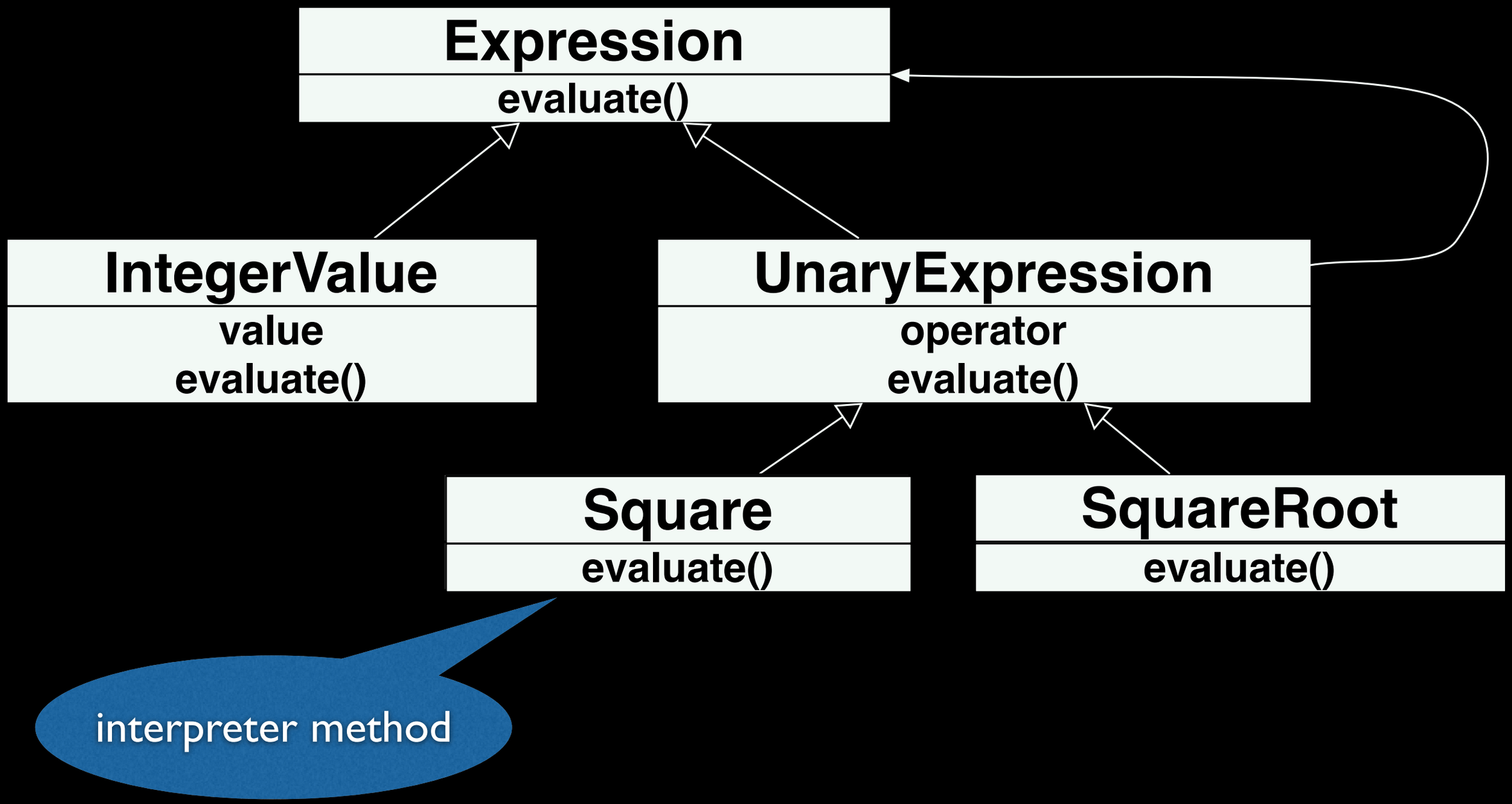

Square
Root
Square
Integer
Value
4


Sum
Integer
Value
1
Square
Root
Square
Integer
Value
4

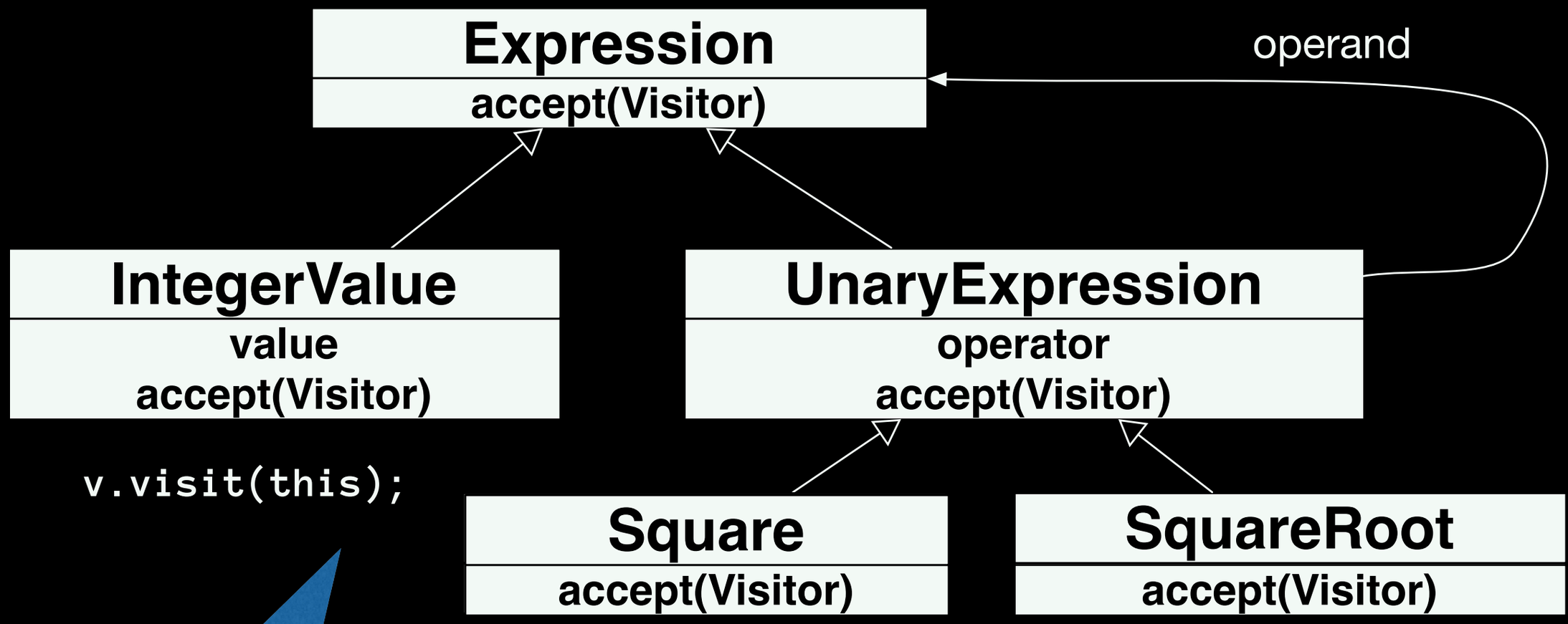
Abstract syntax tree (AST)

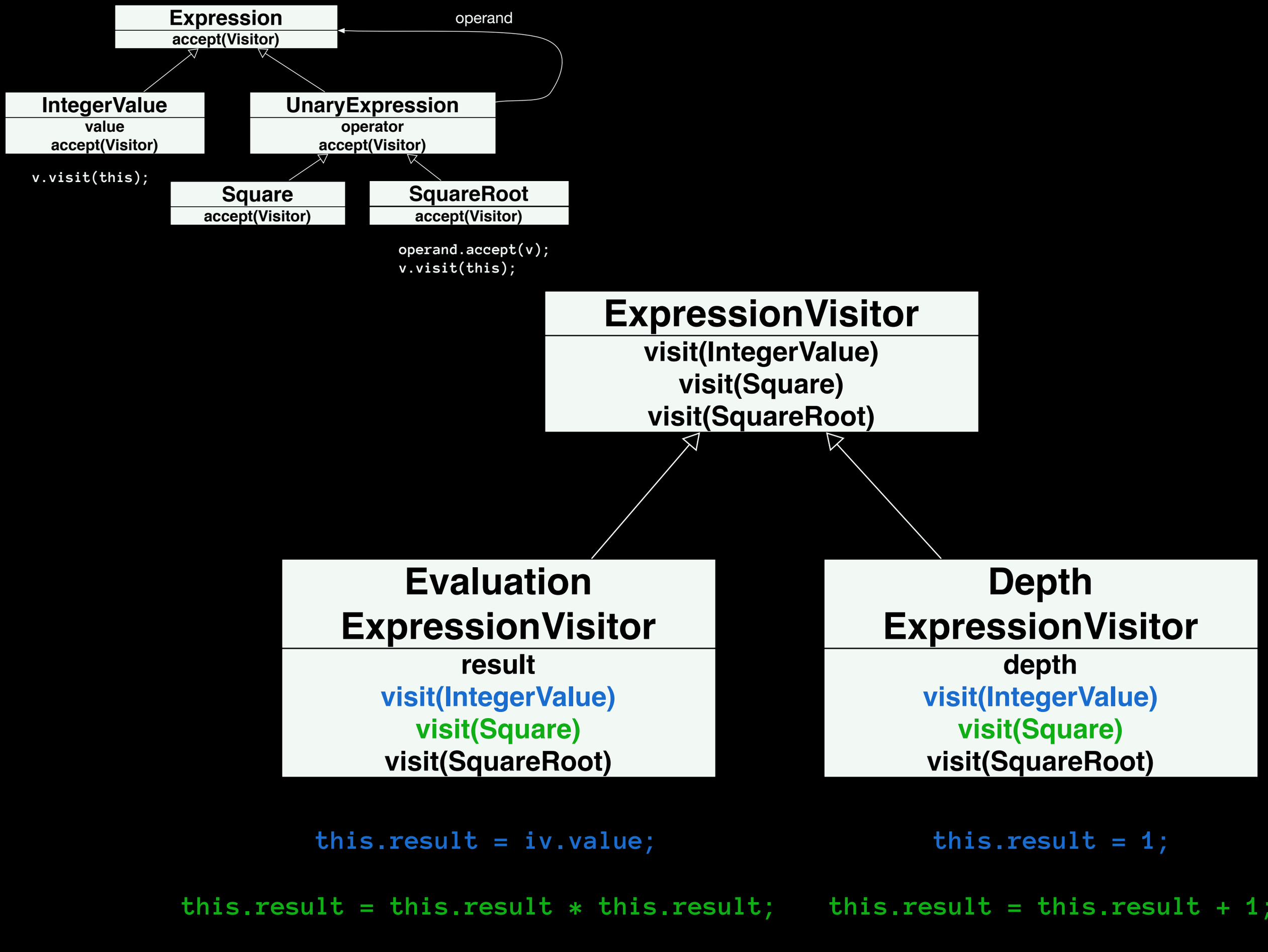


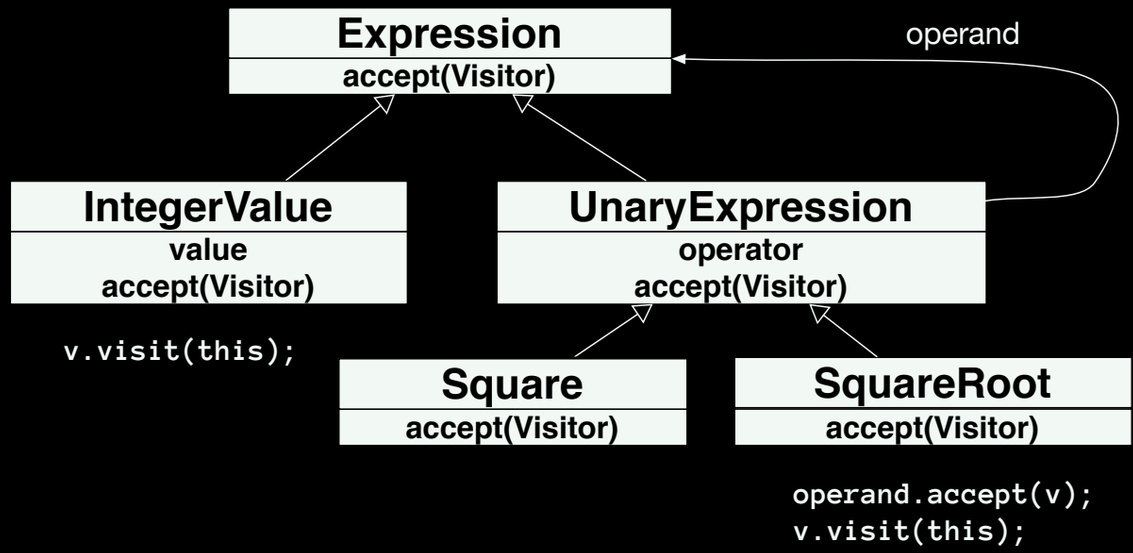
Interpreter

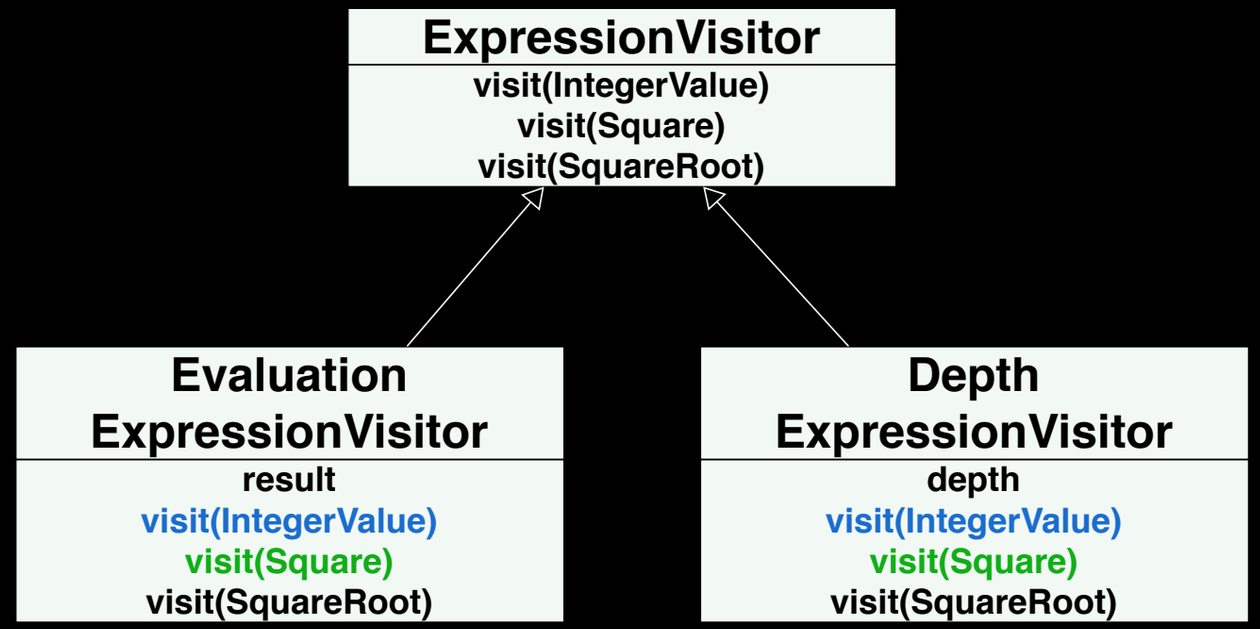


Visitor

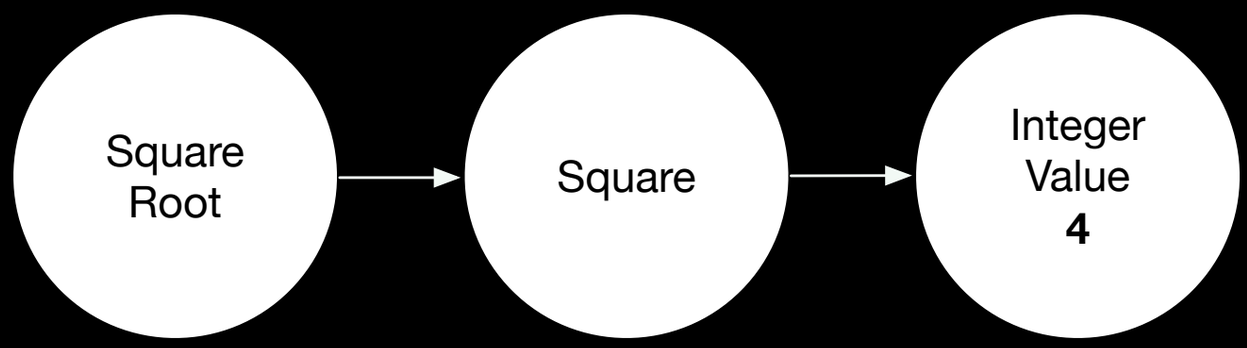

Expression
accept(Visitor)
operand
IntegerValue
value
accept(Visitor)
UnaryExpression
operator
accept(Visitor)
v.visit(this);
Square
accept(Visitor)
SquareRoot
accept(Visitor)
each
visitor provides
different node specific
ofunctionality
operand.accept(v);
v.visit(this);
reusable
traversal logic


Expression
accept(Visitor)
operand
IntegerValue
value
accept(Visitor)
UnaryExpression
operator
accept(Visitor)
v.visit(this);
Square
accept(Visitor)
SquareRoot
accept(Visitor)
operand.accept(v);
v.visit(this);
ExpressionVisitor
visit(IntegerValue)
visit(Square)
visit(SquareRoot)
Evaluation
ExpressionVisitor
result
visit(IntegerValue)
visit(Square)
visit(SquareRoot)
Depth
ExpressionVisitor
depth
visit(IntegerValue)
visit(Square)
visit(SquareRoot)
this.result = iv.value;
this.result = this.result * this.result;
this.result = 1;
this.result = this.result + 1;






```
2. accept(ev);          3. accept(ev);
5. ev.visit(this);      4. ev.visit(this);
```




```
0. ev = new EvaluationExpressionVisitor();
1. accept(ev);
6. ev.visit(this);
```

Other patterns

- Command (better with function, if single function)
-

Strategy

- Analyze change request
- Any pattern for the given context?
- New solution based on values and principles?
- Apply appropriate pattern or principles
- Check whether new functionality breaks existing tests

For now, focus on bad design smells, related to classes and their relationships

Later we will focus on bad code smells, related to method implementations

Applies for testing code too!

Checklist

- Design and implementation conforms to discussed principles and patterns

Take notes,

now!

Hands on exercises

Design, implementation, and maintenance 3

Design, implementation, and maintenance 4

Hands on exercises

Design, implementation, and maintenance 4

Hands on exercises

Design, implementation, and maintenance 5

Hands on exercises

Design, implementation, and maintenance 6

Design, implementation and evolution research at CIn

- Software product lines and evolution: Leopoldo, Paulo
- Formal specification methods: Augusto, Alexandre Mota, Juliano, Henrique, Márcio
- Concurrent systems and green design: Fernando Castor
- Distributed systems architecture: Vinicius, Kiev

To do after class

- Answer questionnaire (check classroom assignment), study correct answers
- Finish exercise (check classroom assignment), study correct answers
- Read, again, chapters 8 and 11 in the textbook
- Evaluate classes (check classroom assignment)
- Study questions from previous exams

To do after class, optional

- Estudar material sobre princípios de projeto
- Estudar princípios de codificação no Capítulo 3 do livro Implementation patterns, Kent Beck, 2008 (slides úteis para ter uma visão geral do livro)
- Navegar pelo catálogo de padrões

Questions from previous exams

- Descreva o princípio "Dependency Injection". Explique as vantagens de seguir este princípio.
- Descreva o objetivo do padrão de projeto "Adapter", e desenhe um diagrama de classes representando a sua estrutura.
- O princípio "Acyclic dependencies"estabelece que "The dependencies between packages must not form cycles". Explique as vantagens de seguir esse princípio.
- Desenhe um diagrama de classes que ilustre o padrão de projeto "Decorator", e explique as suas vantagens.

Questions from previous exams

- Descreva o princípio "Single Responsibility Principle". Explique as vantagens de seguir este princípio.
- Descreva o princípio de projeto "Demeter". Explique as vantagens de seguir este princípio.
- Desenhe um diagrama de classes que ilustre o padrão de projeto "Abstract Factory", e explique que problema ele resolve.

Software and systems engineering

Paulo Borba
Informatics Center
Federal University of Pernambuco

phmb@cin.ufpe.br ◈ twitter.com/pauloborba