Design Patterns

Design Patterns

- Tested, proven and documented solutions for recurring design problems in given contexts.

- Each design pattern is structured as
  - Pattern name
  - Intent
  - Motivation
  - Applicability
  - Class structure
  - Participants
  - …etc.

Resources

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Eric Gamma et al., Addison-Wesley

- *Head First Design Patterns*
  - Elizabeth Freeman et al., O'Reilly

- *Game Programming Patterns*
  - Robert Nystrom, Genever Benning
  - http://gameprogrammingpatterns.com/
- Web
  - http://en.wikipedia.org/wiki/Design_patterns_(computer_science)
  - http://sourcemaking.com/design_patterns

Benefits of Design Patterns

- Useful information source to learn and practice good designs

- Useful as a communication tool among developers
  - c.f. Recursion, collections (array, list, set, map, etc.), sorting, buffers, infinite loops, integer overflow, etc.

Recap: Brief History to OOD

- In good, old days… programs had no structures.
  - One dimensional code.
    - From the first line to the last line on a line-by-line basis.
    - “Go to” statements to control program flows.
      - Produced a lot of “spaghetti” code
        - » “Go to” statements considered harmful.
  - No notion of structures (or modularity)
    - Modularity: Making a chunk of code (module) self-contained and independent from the other code
      - Improve reusability and maintainability
        - » Higher reusability → higher productivity, less production costs
        - » Higher maintainability → higher productivity and quality, less maintenance costs

Modules in SD and OOD

- Modules in Structured Design (SD)
  - Structure = a set of variables (data fields)
  - Function = a block of code
- Modules in OOD
  - Class = a set of data fields and functions
  - Interface = a set of abstract functions
- Key design questions/challenges:
  - how to define modules
  - how to separate a module from others
  - how to let modules interact with each other

SD v.s. OOD

- OOD
  - Intends coarse-grained modularity
    - The size of each code chuck is often bigger.
  - Extensibility in mind in addition to reusability and maintainability
    - How easy (cost effective) to add and revise existing modules (classes and interfaces) to accommodate new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to gain reusability, maintainability and extensibility?
    - Design patterns show good examples.

Suggested Read

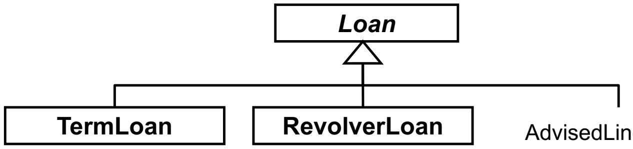
- Chapter 1 (Introduction) of *Game Programming Patterns*
  - http://gameprogrammingpatterns.com/introduction.html

Static Factory Method

Static Factory Method

- Intent
  - Define a “communicable” method to instantiate a class
    - Constructors are the methods to instantiate a class.
    - Static factory methods are more “communicable” (or easier to use) than constructors.
- Benefits
  - Static factory methods have names (!).
    - Improve code maintainability.
      - The name can explicitly tell what object to be created/returned.
      - Client code gets easier to understand.

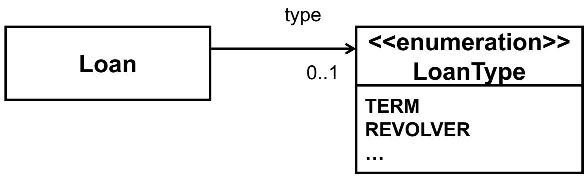
Recap: This Design is not Good.



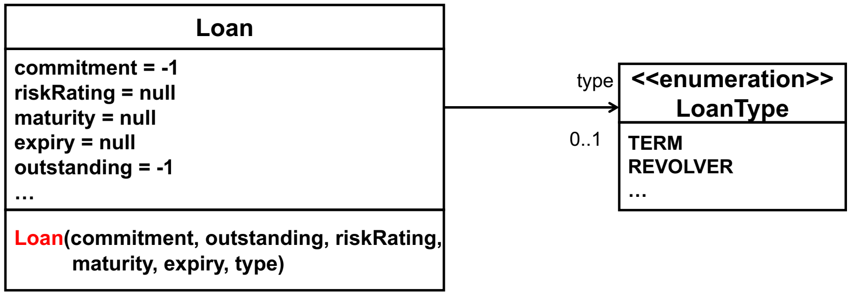
AdvisedLineLoan
SPLC
RCTL
risk adjusted versions
etc.

- Term loan
  - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
  - With a spending limit and expiry date
- Dynamic class change problem
  - A revolver can transform into a term loan when the revolver expires.

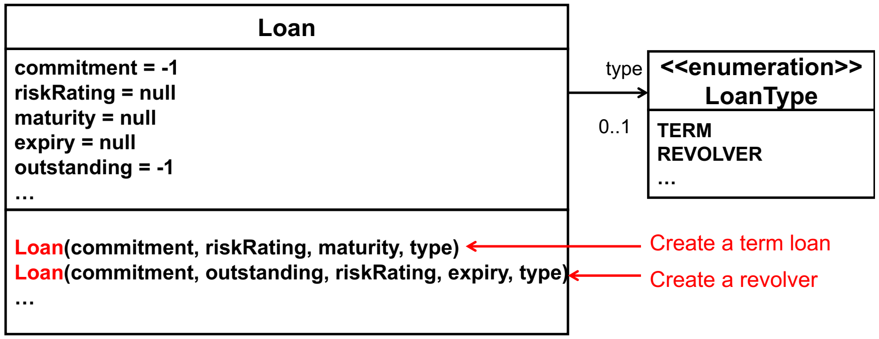
Enumeration-based Design



- A class inheritance should not be used here.

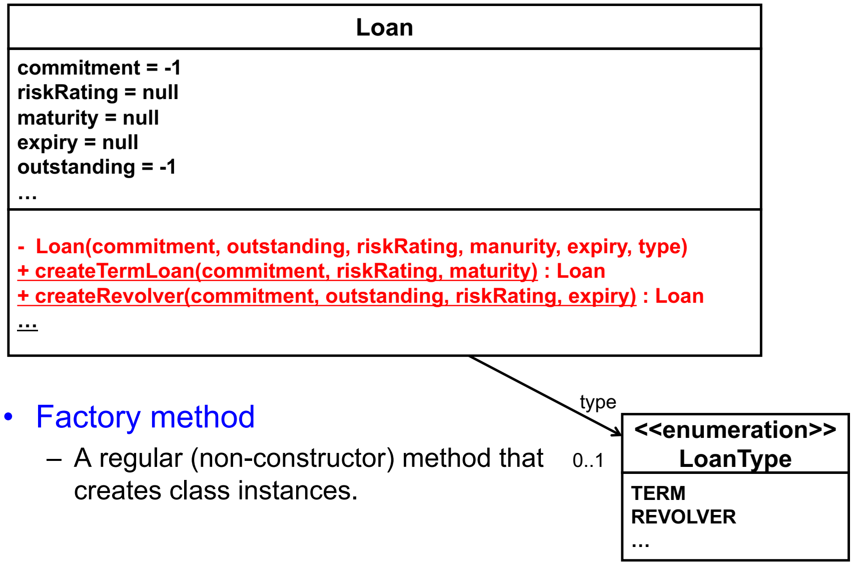


- Different loans need different sets of data to be set up.
  - A term loan needs commitment, risk rating and maturity date.
  - A revolver needs commitment, outstanding debt, risk rating and expiry date.
- The constructor is error-prone. Its client code is hard to understand/maintain.
  - `Loan l1 = new Loan(100, -1, 0.9, new Date(…), null, LoanType.TERM);`
  - `Loan l2 = new Loan(100, 0, 0.7, null, new Date(…), LoanType.REVOLVER);`



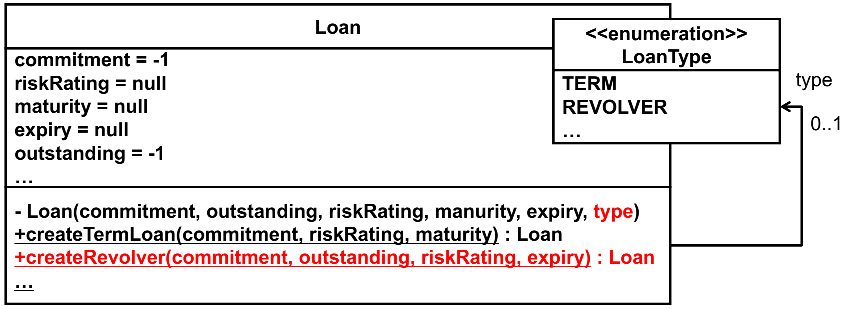
- Multiple constructors to create different types of loans.
- They are less error-prone, their client code is still hard to understand/maintain.
  - `Loan l1 = new Loan(100, 0.9, new Date(…), LoanType.TERM);`
  - `Loan l2 = new Loan(100, 0, 0.7, new Date(…), LoanType.REVOLVER);`

Static Factory Methods



Loan
commitment = -1
riskRating = null
maturity = null
expiry = null
outstanding = -1
…
- Loan(commitment, outstanding, riskRating, manurity, expiry, type)
+ createTermLoan(commitment, riskRating, maturity) : Loan
+ createRevolver(commitment, outstanding, riskRating, expiry) : Loan
…
type
0..1
<<enumeration>>
LoanType
TERM
REVOLVER
…

- Factory method
  - A regular (non-constructor) method that creates class instances.



Loan
commitment = -1
riskRating = null
maturity = null
expiry = null
outstanding = -1
…
- Loan(commitment, outstanding, riskRating, manurity, expiry, type)
+createTermLoan(commitment, riskRating, maturity) : Loan
+createRevolver(commitment, outstanding, riskRating, expiry) : Loan
…
<<enumeration>>
LoanType
TERM
REVOLVER
…
type
0..1

- Client/user of Loan
  - `Loan loan = Loan.createRevolver(1000,0,…,…);`

```
Public class Loan{
  private LoanType type = null;
  ……
  private Loan(…,…,…,…,…,…){ … }
  public static Loan createRevolver( commitment, outstanding,
                                    riskRating, expiry ){
    return new Loan( commitment, outstanding, riskRating, null,
                     expiry, LoanType.REVOLVER );
  }
```

- You should not define public constructors.
- If you do not define constructors…

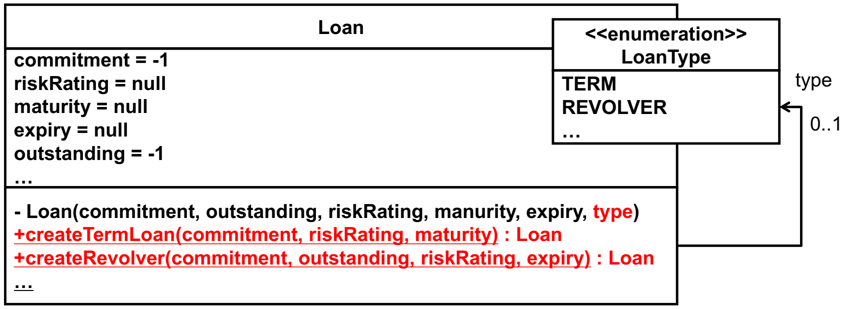
Benefits of *Static Factory Method*

- Static factory methods have their own names.
  - Improve code maintainability.
    - The name can explicitly tell what object is created and what data is required to set it up.
    - Client code gets clean and easier to understand.

```
– Loan l1 = new Loan(100, 0.9, new Date(…), LoanType.TERM);

– Loan l2 = Loan.createTermLoan(100, 0.9, new Date(…));
```

A Potential Issue w/ *Static Factory Method*



Loan

commitment = -1
riskRating = null
maturity = null
expiry = null
outstanding = -1
…

- Loan(commitment, outstanding, riskRating, manurity, expiry, type)
+createTermLoan(commitment, riskRating, maturity) : Loan
+createRevolver(commitment, outstanding, riskRating, expiry) : Loan
…

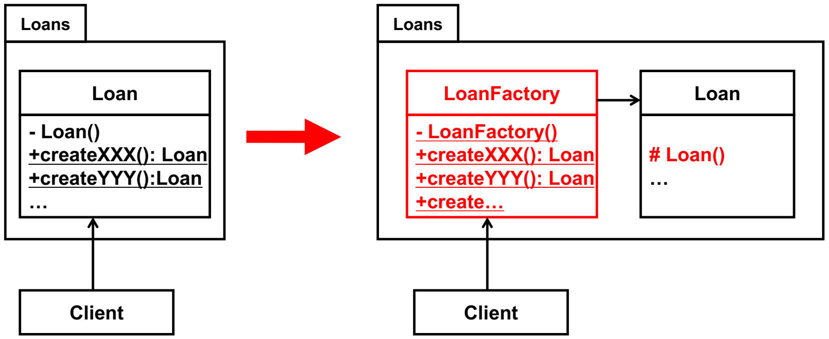
<<enumeration>>
LoanType

TERM
REVOLVER
…

type
0..1

- Too many (static) factory methods in a class may obscure its primary responsibility/functionality.
  - They may dominate the class's public methods.
  - Loan may no longer strongly communicate it's primary (i.e., loan-related) responsibility/functionality.

Alternative: Factory Class



Loans

Loan

- Loan()
+createXXX(): Loan
+createYYY():Loan
…

Client

Loans

LoanFactory

- LoanFactory()
+createXXX(): Loan
+createYYY(): Loan
+create…

Loan

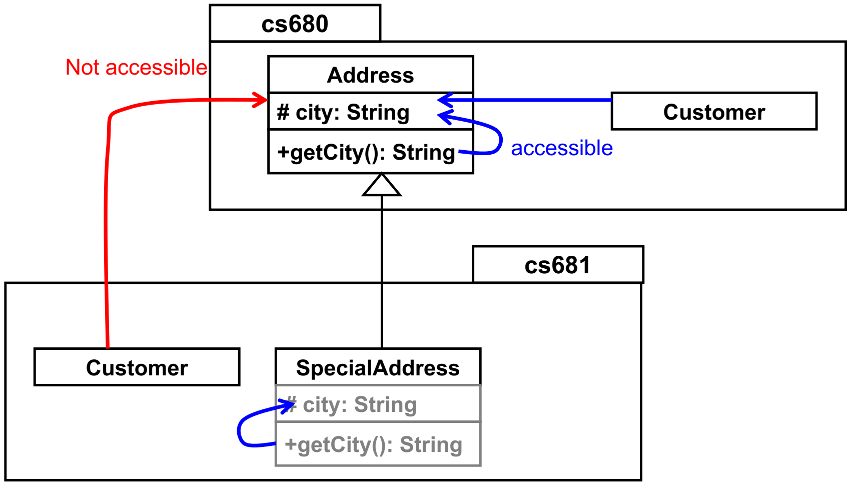
# Loan()
…

Client

- Factory class
  - A class that encapsulates static factory methods and isolate instantiation logic from other classes.

- Loan can now directly/strongly communicate its primary responsibility/functionality.

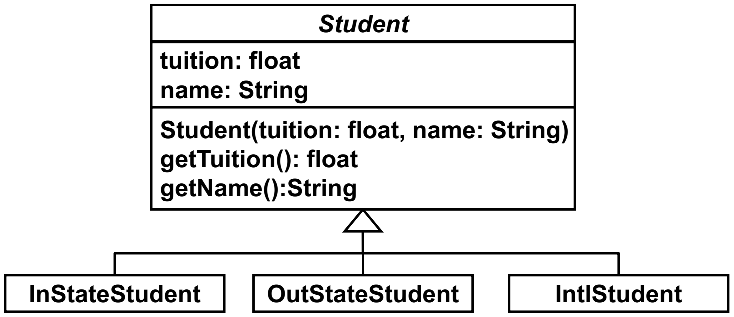
Recap: "Protected" Visibility



Design Tradeoffs

- Class inheritance
  - Pros: Straightforward.
  - Cons: Dynamic class change is hard (virtually impossible) to implement.
- Static factory method
  - Pros: Can avoid dynamic class change problem.
  - Cons: Potentially too many factory methods in a single class
- Factory class
  - Pros: Separates a class's primary logic and its instantiation logic
  - Cons: Non-factory classes in the same package can call protected constructors.
    - Could violate the encapsulation principle.
    - Consider an inner class (e.g., Loan as an inner class of LoanFactory)

Recap: This Design is not Good.



Student

tuition: float
name: String

Student(tuition: float, name: String)
getTuition(): float
getName():String

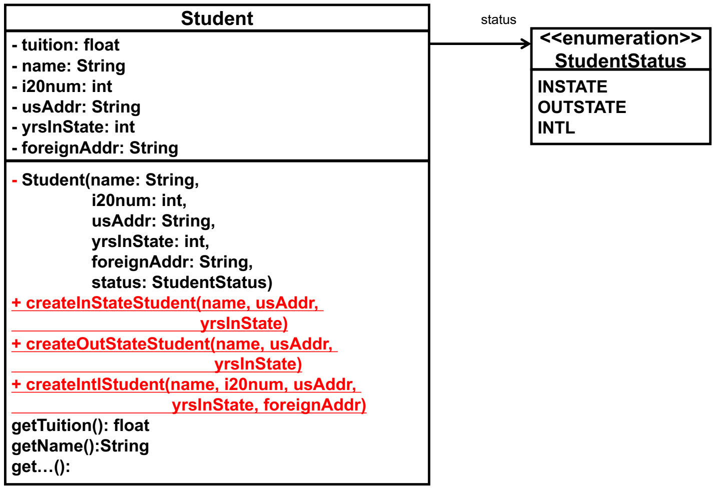
InStateStudent

OutStateStudent

IntlStudent

- Alternative designs
  - Use an enumeration
  - Use *Static Factory Method* and an enumeration

Design Improvement w/ *Static Factory Method*



Student

- tuition: float
- name: String
- i20num: int
- usAddr: String
- yrsInState: int
- foreignAddr: String

```
- Student(name: String,
          i20num: int,
          usAddr: String,
          yrsInState: int,
          foreignAddr: String,
          status: StudentStatus)
+ createInStateStudent(name, usAddr,
                       yrsInState)
+ createOutStateStudent(name, usAddr,
                        yrsInState)
+ createIntlStudent(name, i20num, usAddr,
                    yrsInState, foreignAddr)
getTuition(): float
getName():String
get…():
```

status

<<enumeration>>
StudentStatus

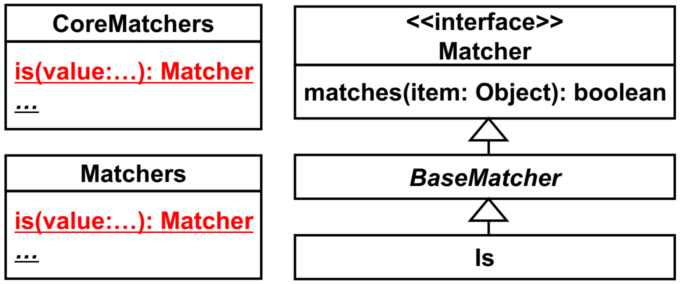
INSTATE
OUTSTATE
INTL

HW 2

- Complete the **`Student`** class and test its (3) static factory methods.
  - Write test cases with JUnit.
  - Optional: Implement a factory class
    - Separate **`StudentFactory`** and **`Student`**
- Deadline: March 6 (Tue) midnight

Another Example: Matchers in Hamcrest

- **`org.hamcrest.CoreMatchers`**
- **`org.hamcrest.Matchers`**
  - Contains static methods, each returning a matcher object that performs matching logic.
  - **`Matchers`** is a superset of **`CoreMatchers`**.
- **`is()`** is a static factory method that creates an instance of **`Is`**.



CoreMatchers

is(value:…): Matcher

…

Matchers

is(value:…): Matcher

…

<<interface>>
Matcher

matches(item: Object): boolean

BaseMatcher

Is

Suggested Read

- Chapter 2 (Creating and destroying Objects) of *Effective Java*
  - Joshua Bloch, Addison
  - http://bit.ly/2ydblp8

Singleton Design Pattern

Singleton

- Intent
  - Guarantee that a class has only one instance.

```
public class Singleton{
    private Singleton(){};
    private static Singleton instance = null;

    public static Singleton getInstance(){
           if(instance==null)
                  instance = new Singleton ();
           return instance;
    }
}
```

- You should not define public constructors.
- If you do not define constructors...

```
Singleton instance = Singleton.getInstance();
instance.hashCode();
Singleton instance = Singleton.getInstance();
instance.hashCode();
```

- hashCode() returns a unique ID for each instance.
  - Different instances of a class have different IDs.

- *Singleton* is an application of *Static Factory Method*.
  - getInstance() is a static factory method.
  - *Singleton* focuses on a requirement to have a class keep only one instance.

What Can be a Singleton?

- Object pools
- Logger
- Plugin manager
- Access counter
- Game loop
- …, etc.