

CS525 – ASD – Advanced Software Design

Class notes and diagrams

These are simple (alpha) versions of in-class drawings and illustrations, presented here only for personal use of students in the summer 2013 ASD on-campus class. They do not completely cover the lecture or lesson content, and are intended only as an aid for review of the class lessons.

You should use them as a supplement to your own class notes, and the online *Pattern Summary* notes and other materials from the course LMS site. Specifically various printed materials used in class are not included here, since they are already available in these other sources.

This is a first version, and only partially complete so far, I will update as we move through the course if you find it useful.

This is **only** for your own personal private usage, it is not to be copied or further distributed in any way because it is just a very preliminary version. This is a work-in-progress document in response to student requests for the in-class diagrams I use in the lessons; I hope you find it useful.

Dr. Guthrie

Sep. 2015

Lessons:

[Day 1: Introduction and Overview](#)

[Day 2: OO programming Model and laws of Software & Design](#)

[Day 3: Pattern Mining & Swing, Observer Pattern](#)

[Day 4: Façade, Strategy, and Template Patterns](#)

[Day 5: Iterator Pattern](#)

[Day 6: More Iterator s](#)

[Day 7a: Functional Iterators & FP](#)

[Day 7b: Composite Pattern](#)

[Day 8: Command & Mediator Patterns](#)

[Day 9: More Command Patterns](#)

[Day 10: State Pattern](#)

[Day 11: Proxy & COR](#)

[Day 12: Dynamic & Generic Proxy](#)

[Day 13: Dynamic Proxy](#)

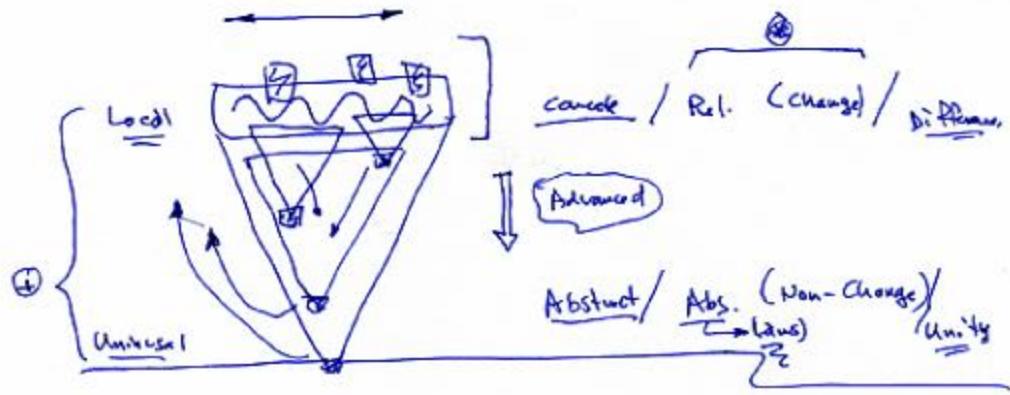
[Day 13b: Factory Pattern](#)

[Day 14: Project Assignment](#)

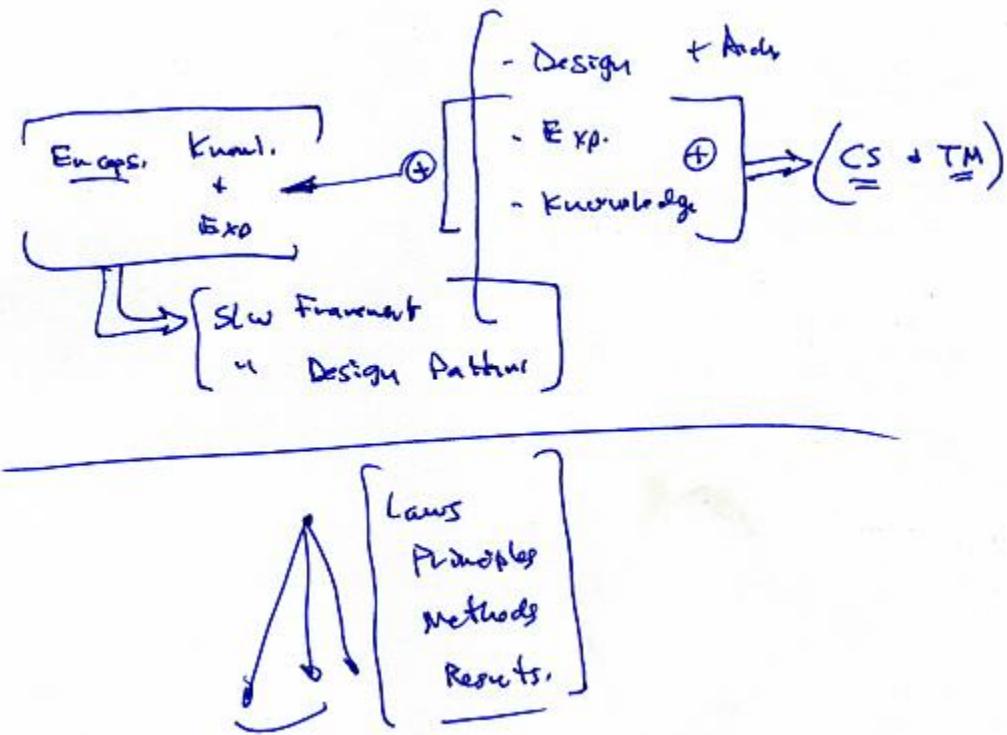
Day 1: Introduction and Overview.

Day 1: Introduction and course Overview

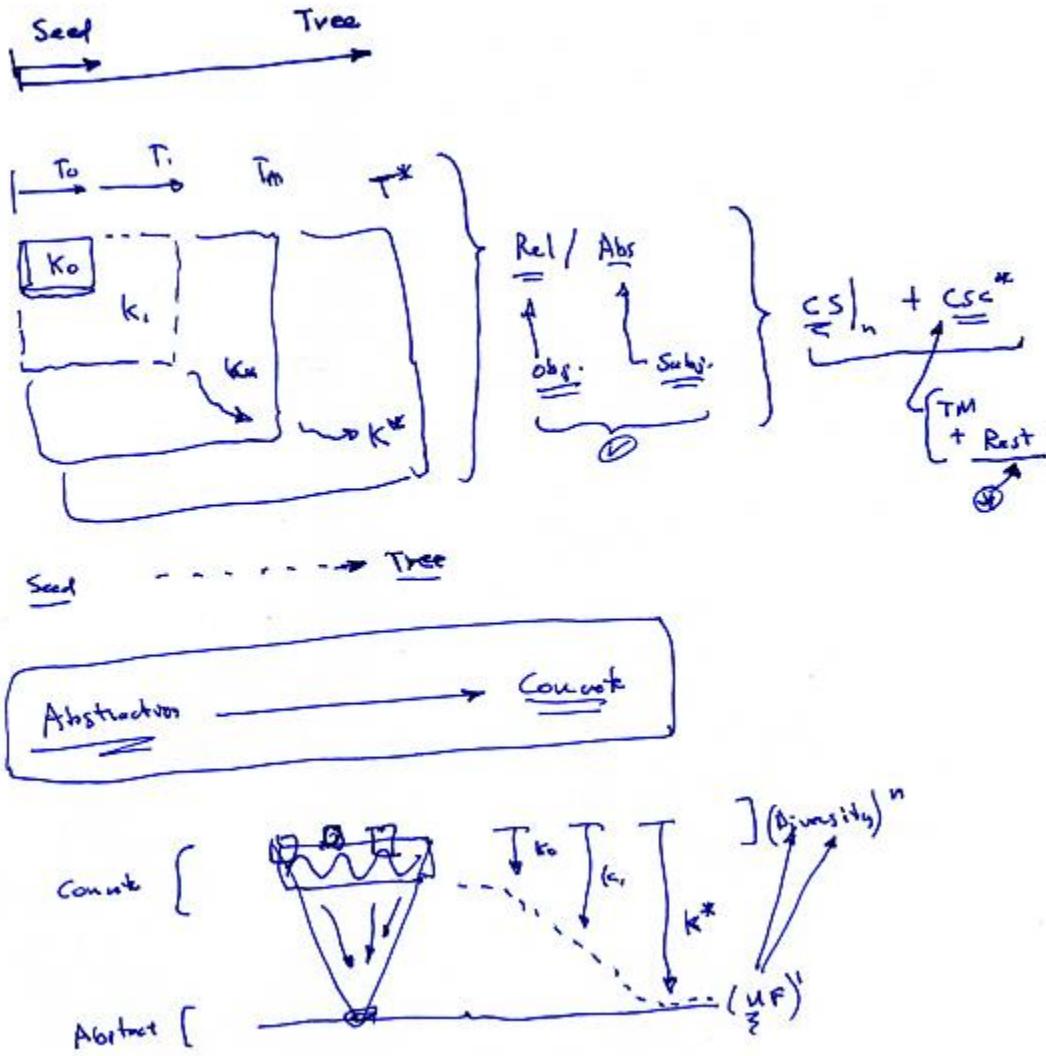
- Overview of Advanced Software Systems
 - Value of Abstraction
 - Elements of Software Development
- Definition of Advanced Software Development & topics
 - Methods of ASD
 - Refactoring and SE⁺⁺
- Course Structure & Administration
 - Daily Schedule
 - Quizzes
 - Labs:
 - *Timelog*, hand-in times
 - Grading, Exams
- Levels of architecture
 - PLangs, idiom, patterns, Frameworks
- Software Patterns & Frameworks; overview
 - Patterns, and design
 - GOF & pattern formalizations
 - Frameworks
- Software Metrics \Leftrightarrow Laws \Leftrightarrow Principles
- Patterns are both language { independent, relative } (?)!
- Reading
 - Homework



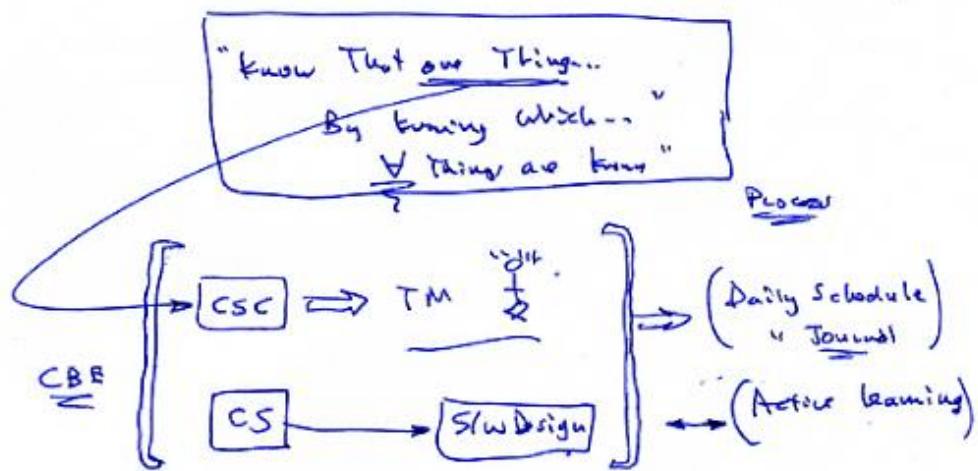
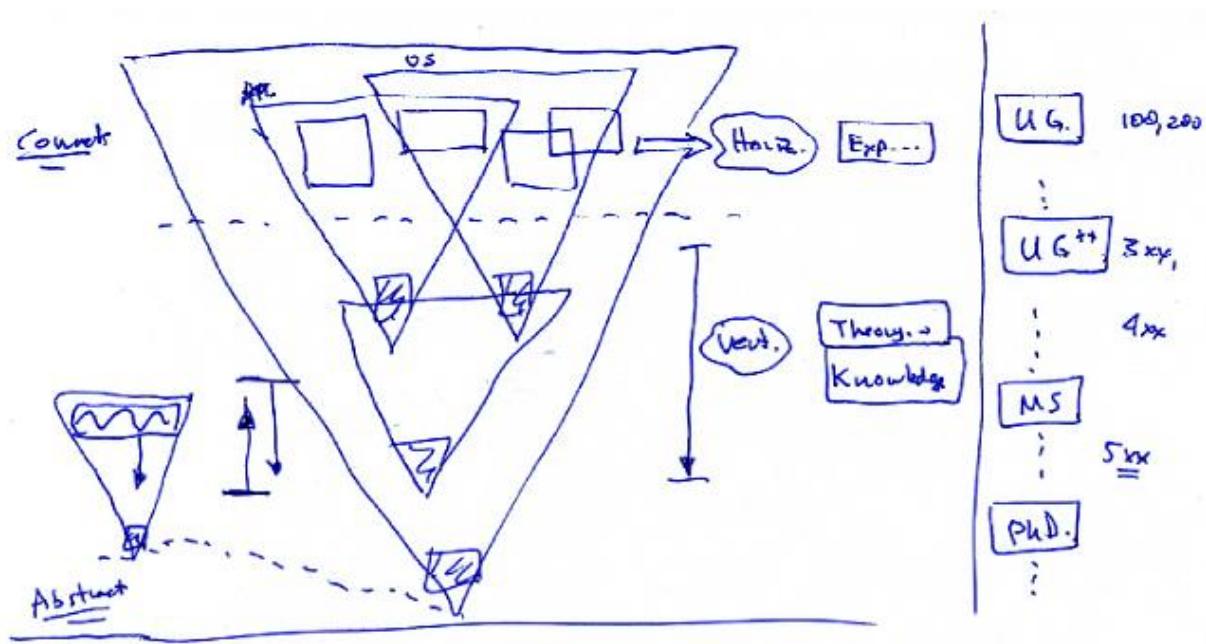
- Levels of abstraction



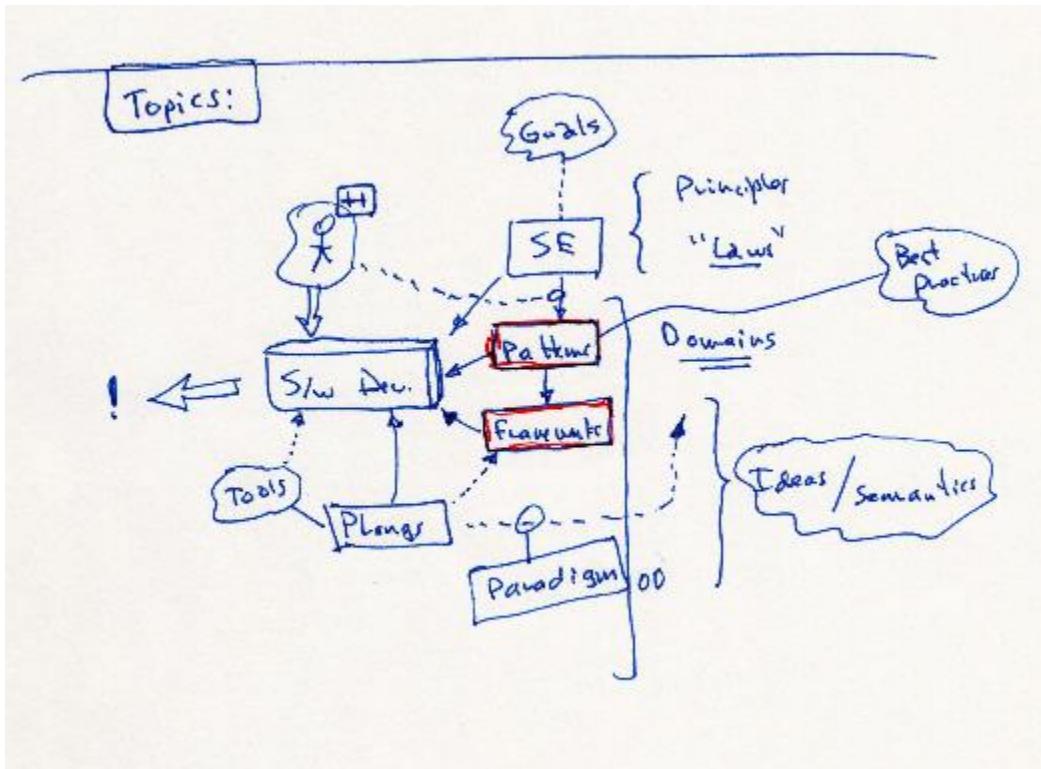
- The meaning of Advanced Knowledge



- Levels of Knowledge and Abstraction, from Abstract to concrete, from wholeness to details of expression.



- The importance of Consciousness



- Course Topics

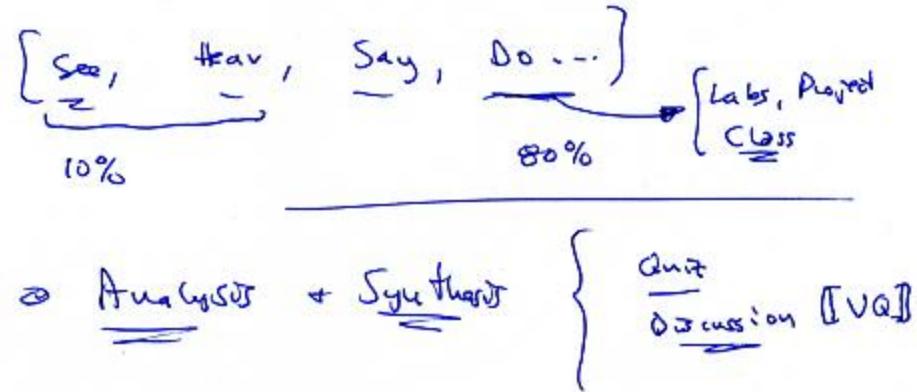
Optimal Learning:

- o Active - Exercise / Q/A - $\approx 80\%$
- Notes..
- o Passive - (ZTV) - $5\% - 20\%$

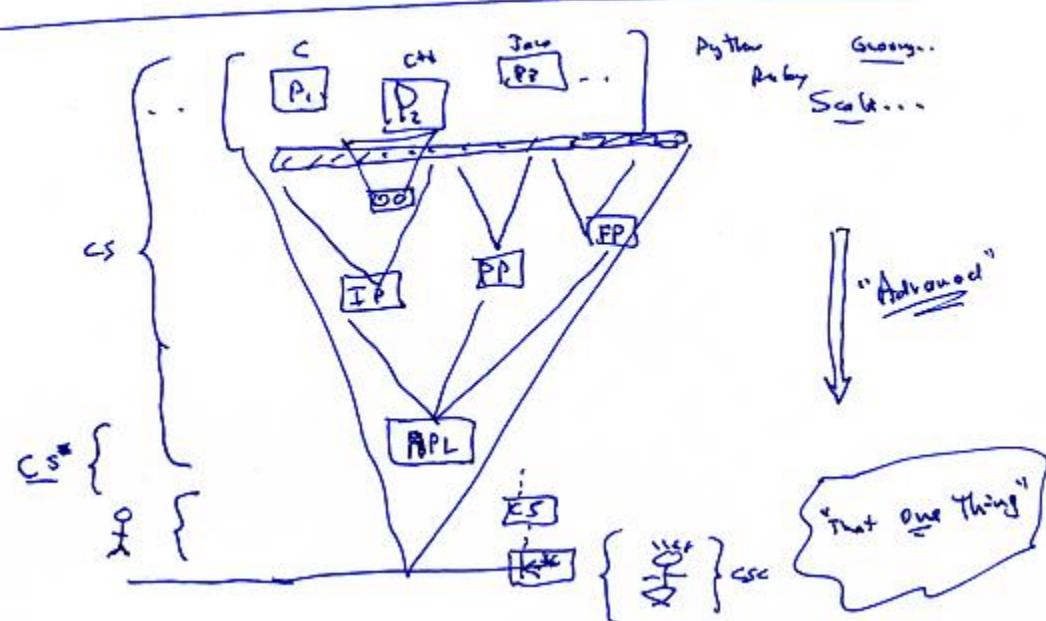
o "Virtuel"

- o Surface / Deep
- o \oplus

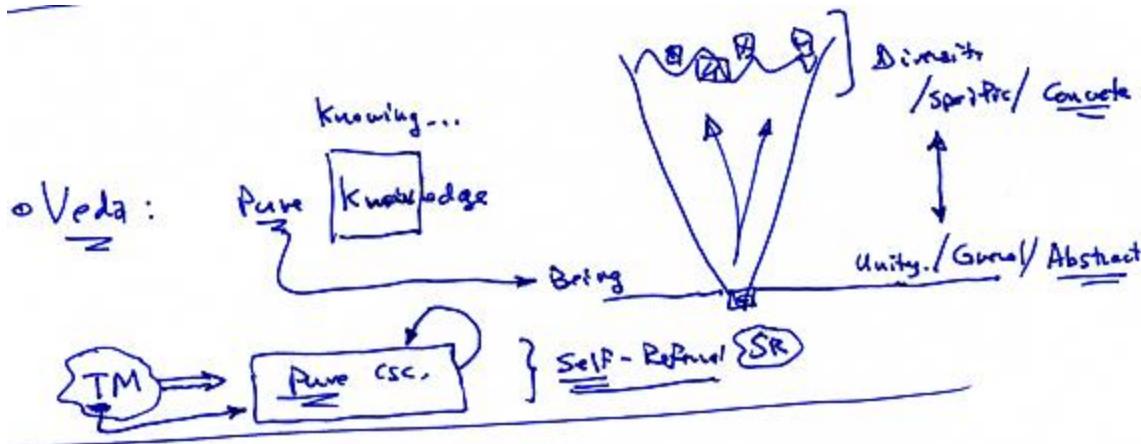
- Active Learning



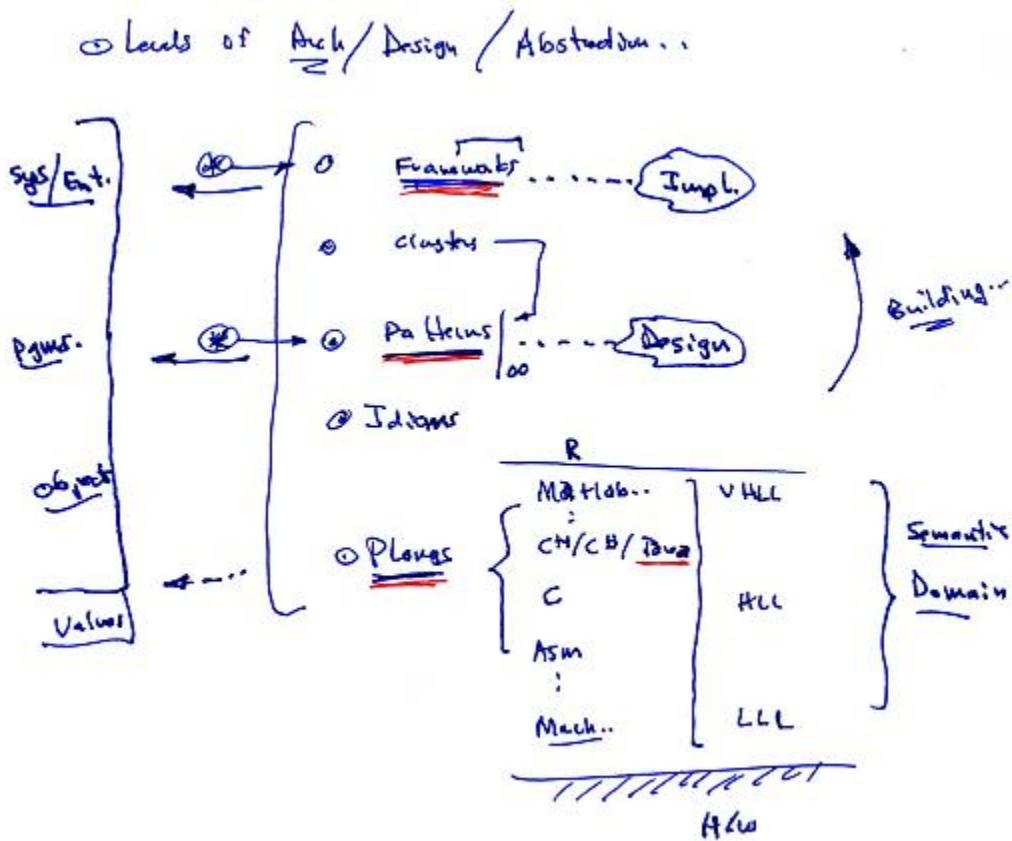
- Active Learning
- The importance of a good daily schedule



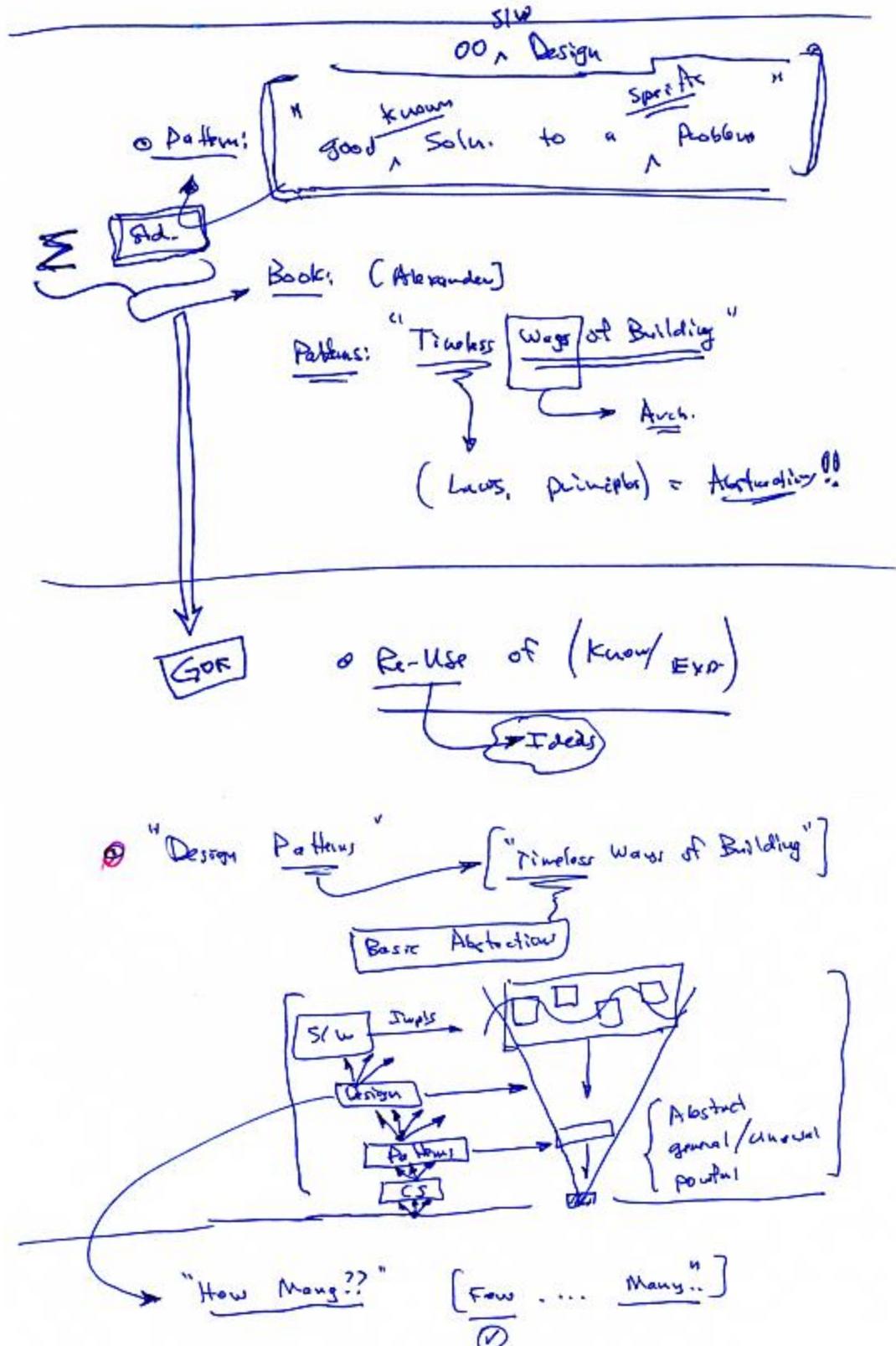
- Levels of Abstraction
- More abstract \Rightarrow more general, more universal, more powerful



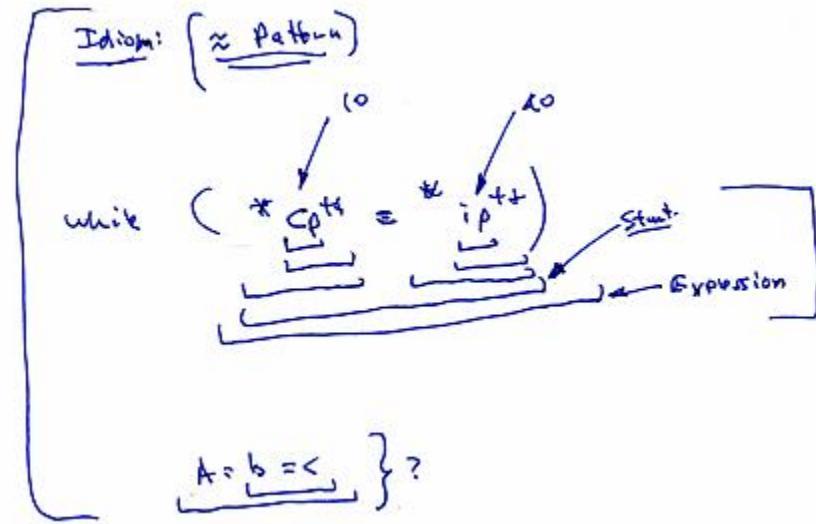
- Levels of Knowledge; Veda \equiv Total Knowledge



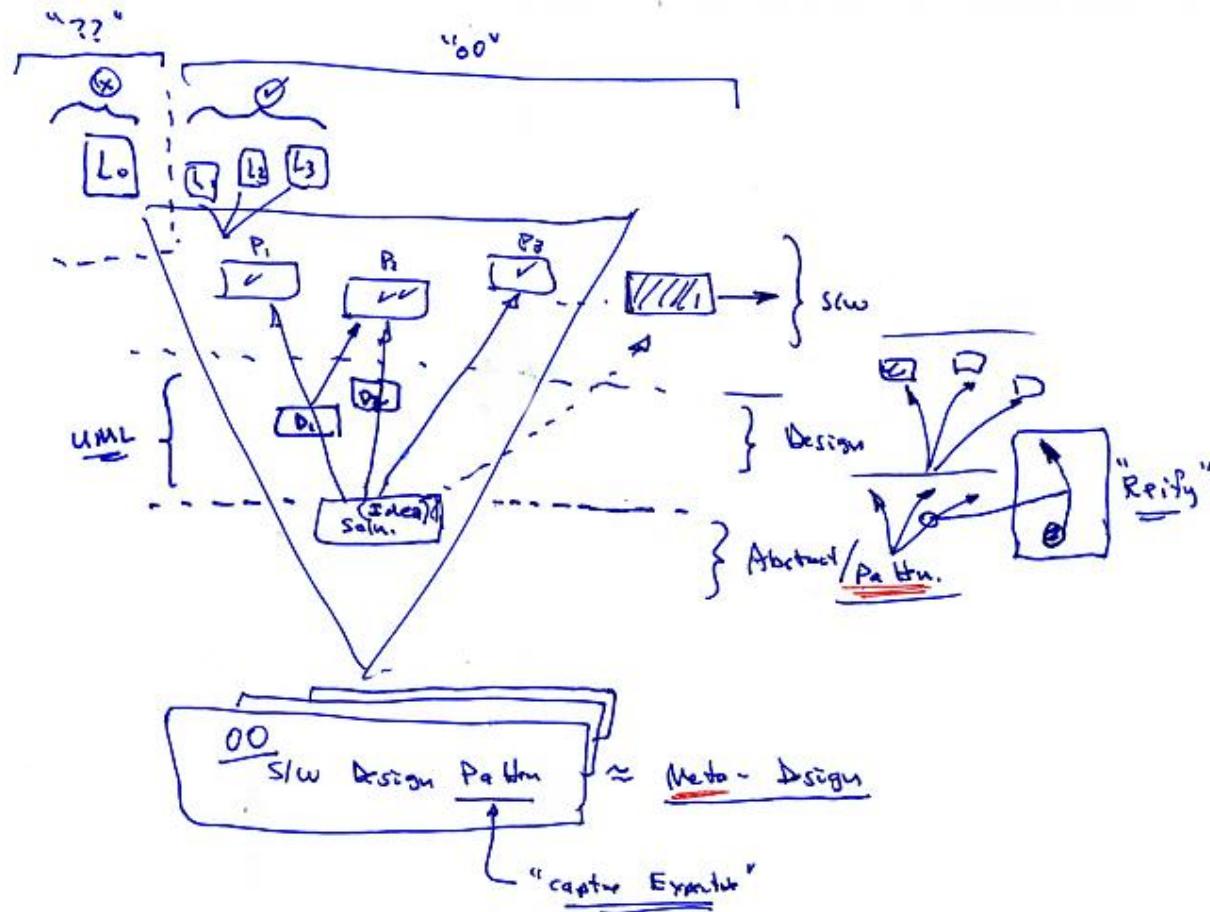
- Levels of Architecture & Design



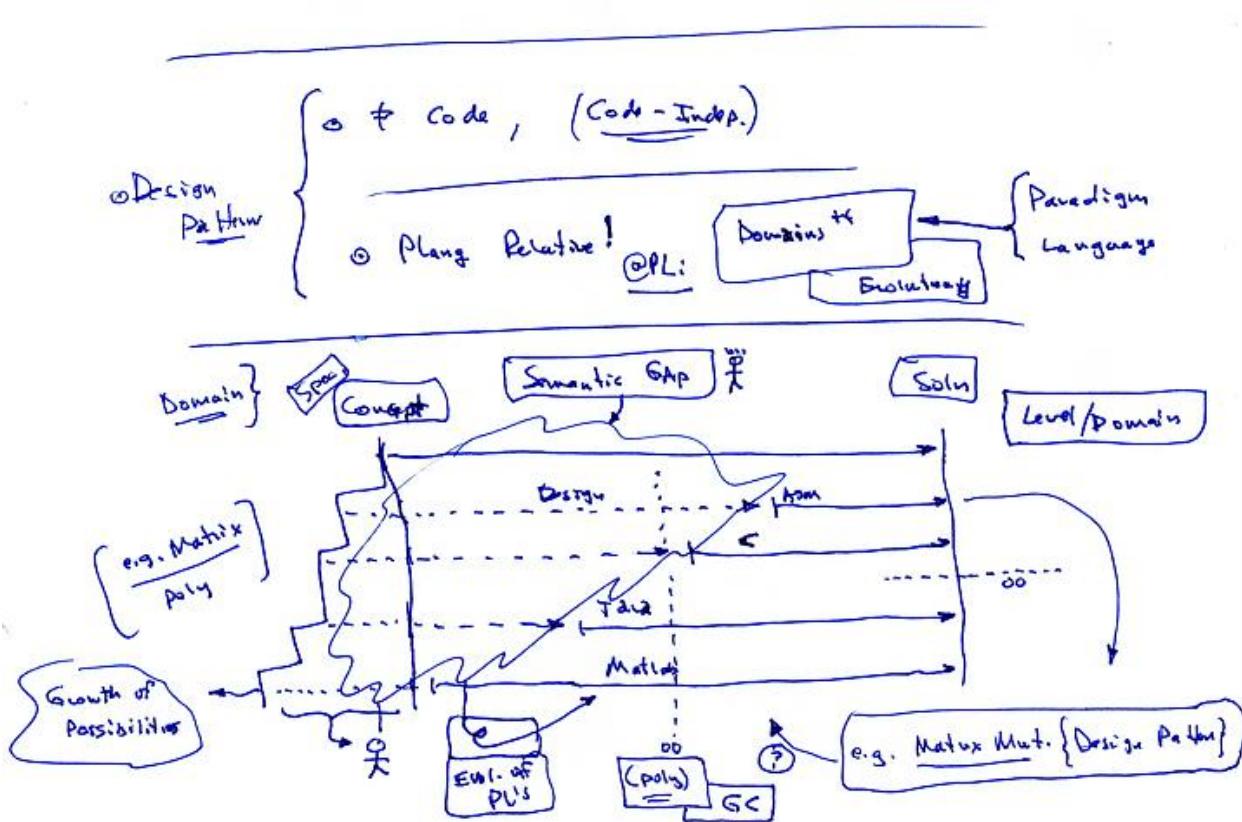
- Definition of Patterns



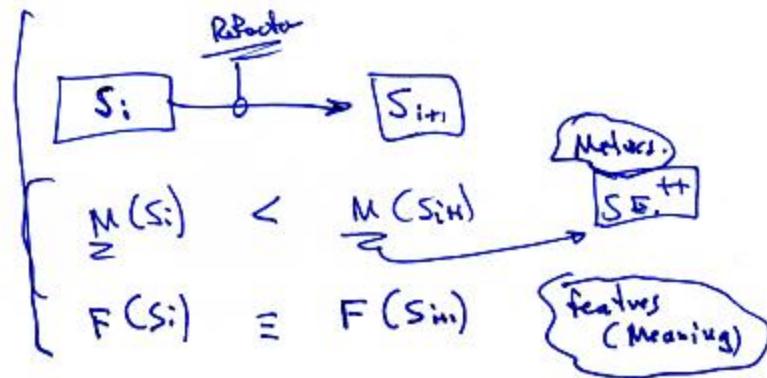
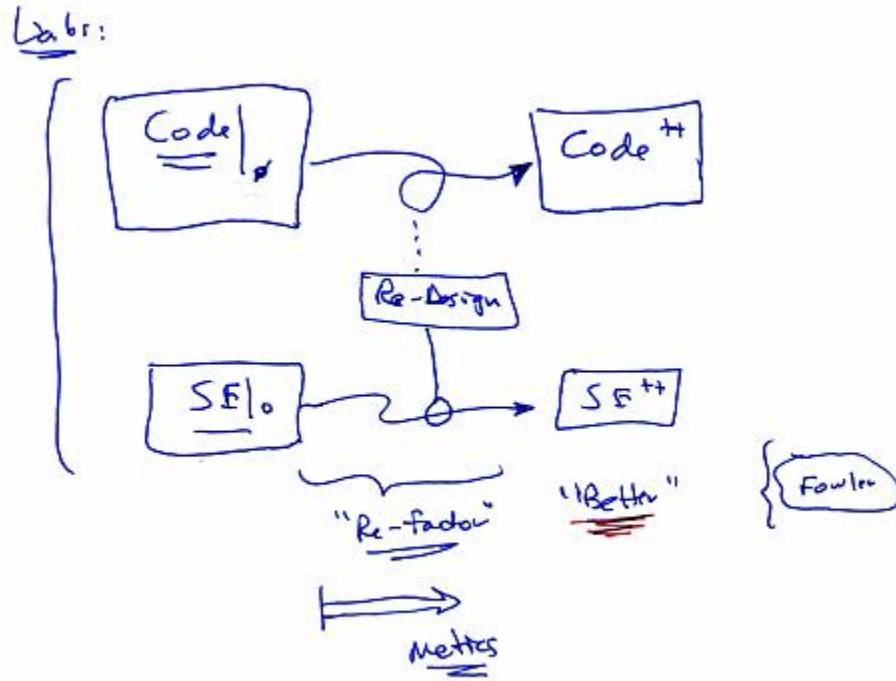
- Idiom = Mini-patterns



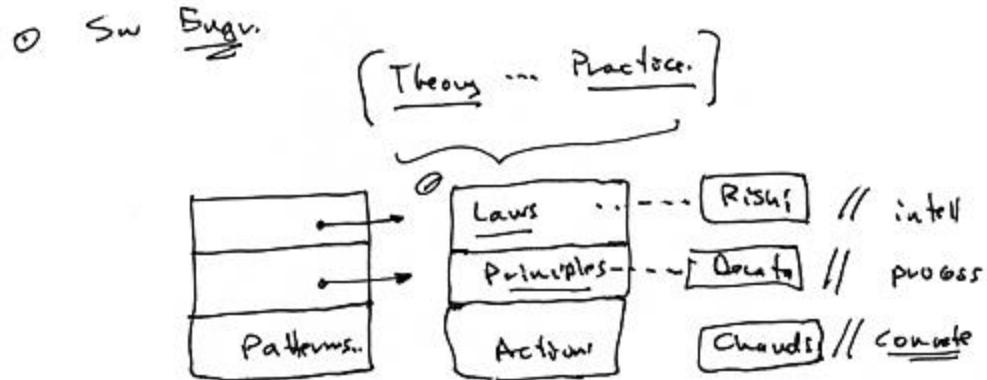
- Patterns are more abstract than designs
- “*Meta-Designs*”
- Patterns *reify* to designs



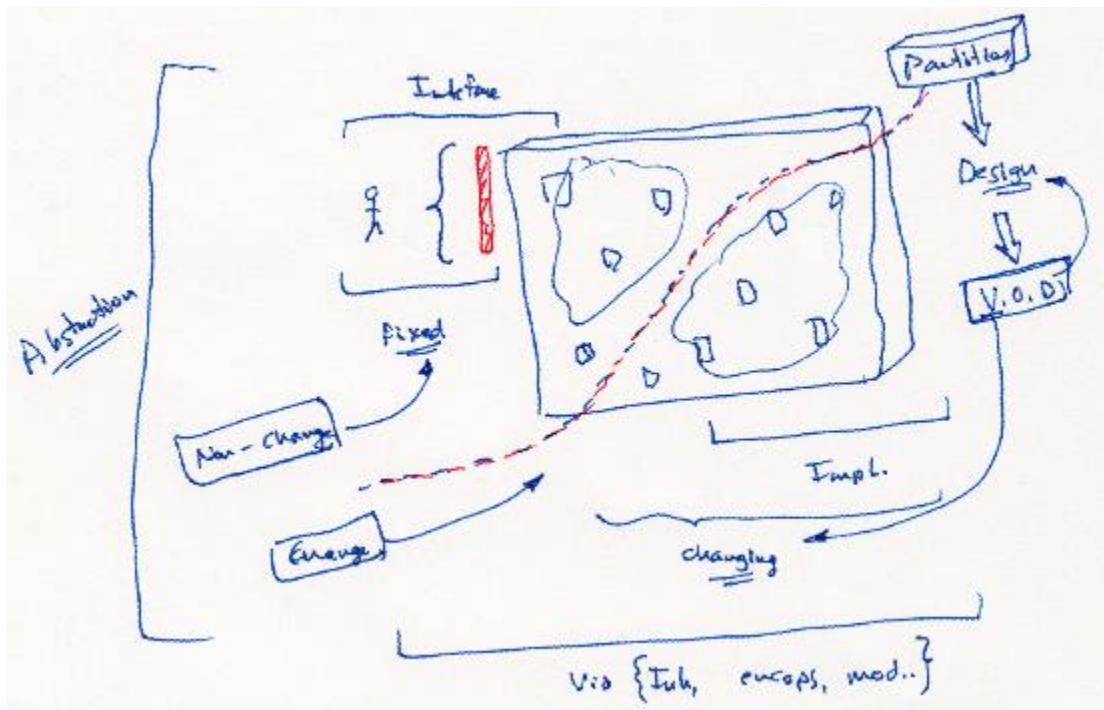
- Relationship of Patterns and Programming languages



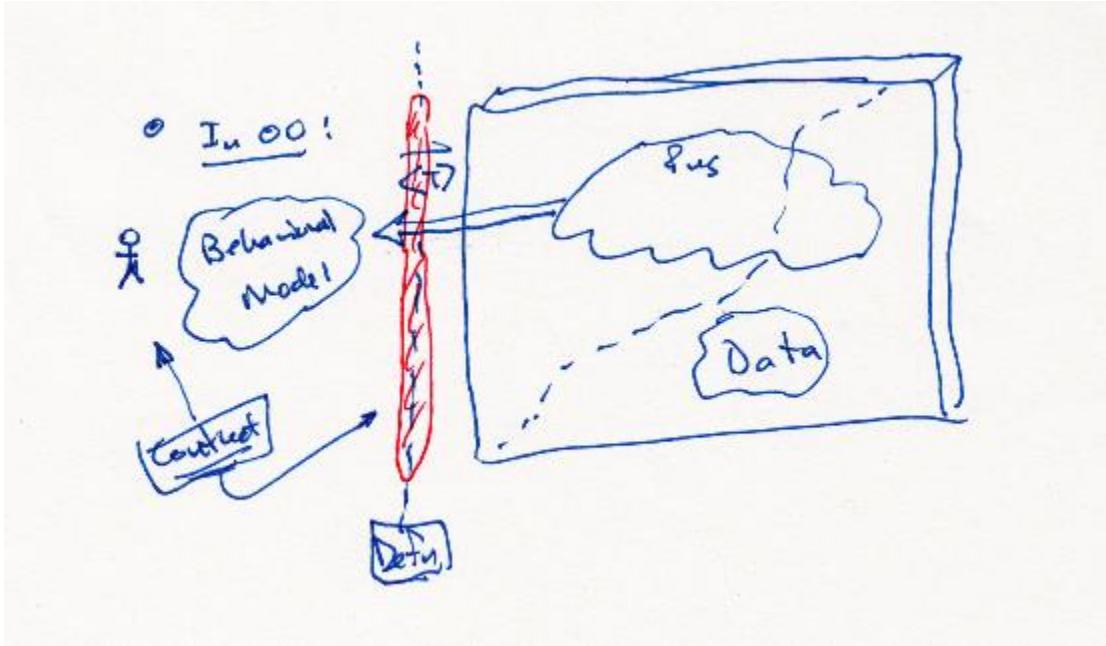
- Refactoring



- Laws and Principles of SE & Design

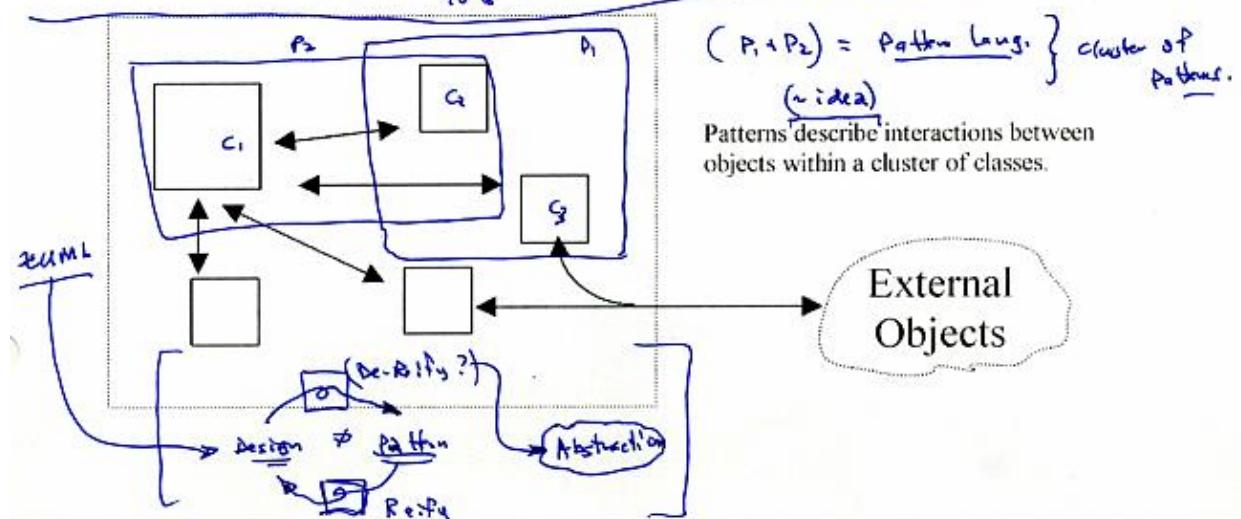
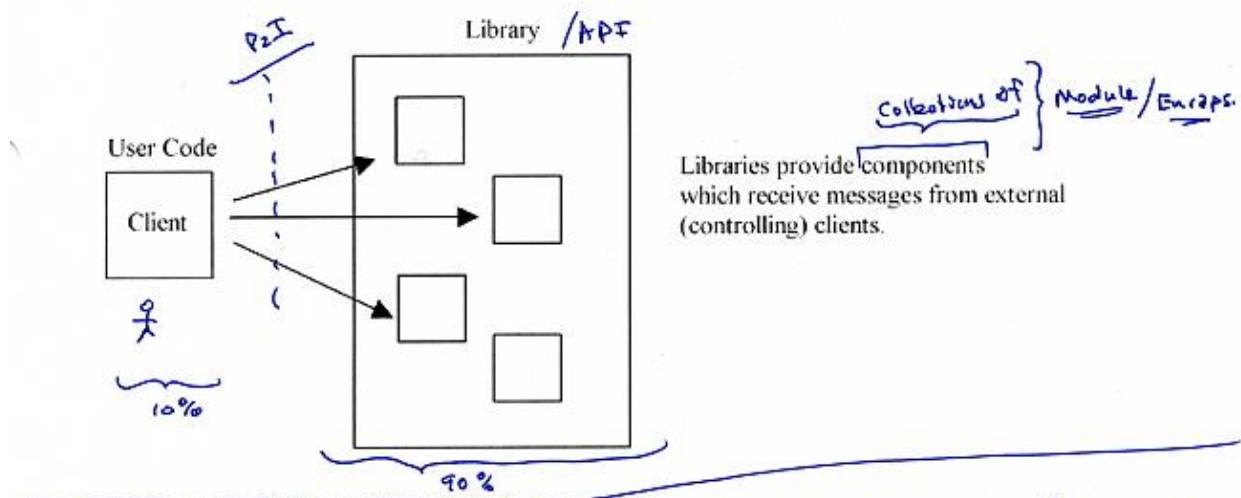
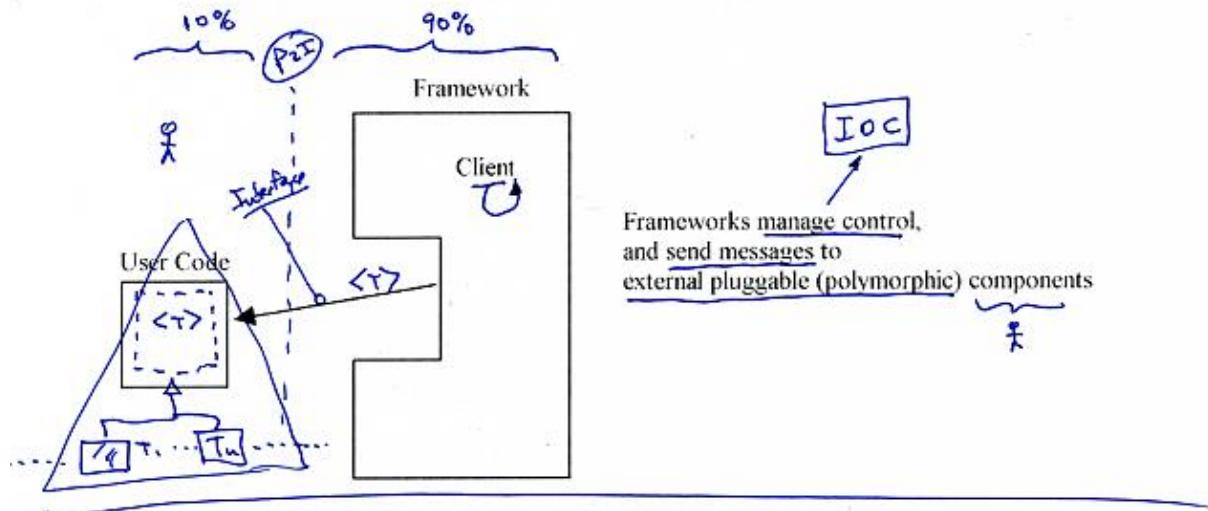


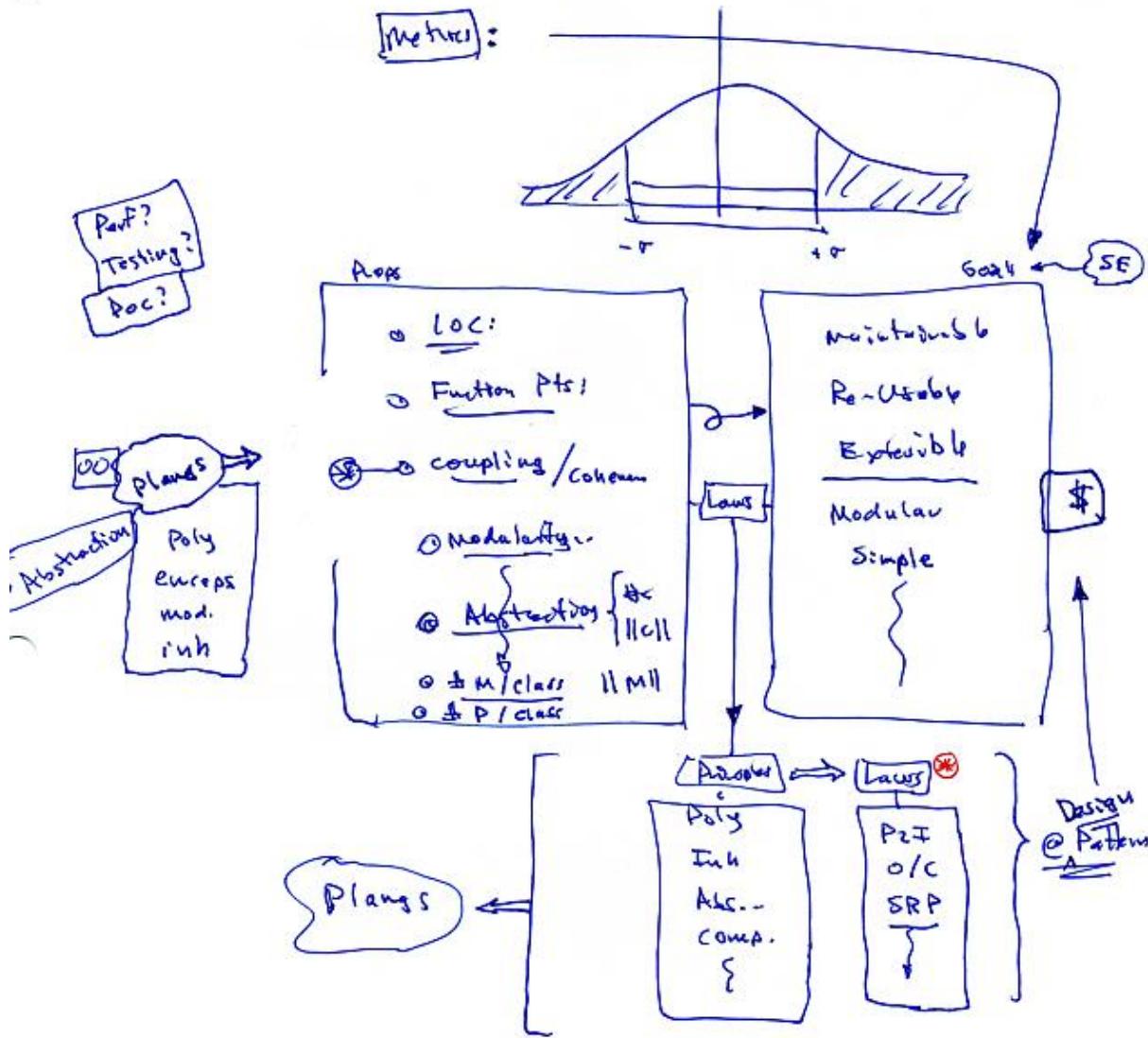
- VOD – Variation Oriented Design – means partitioning a system into two parts; change and non-change.



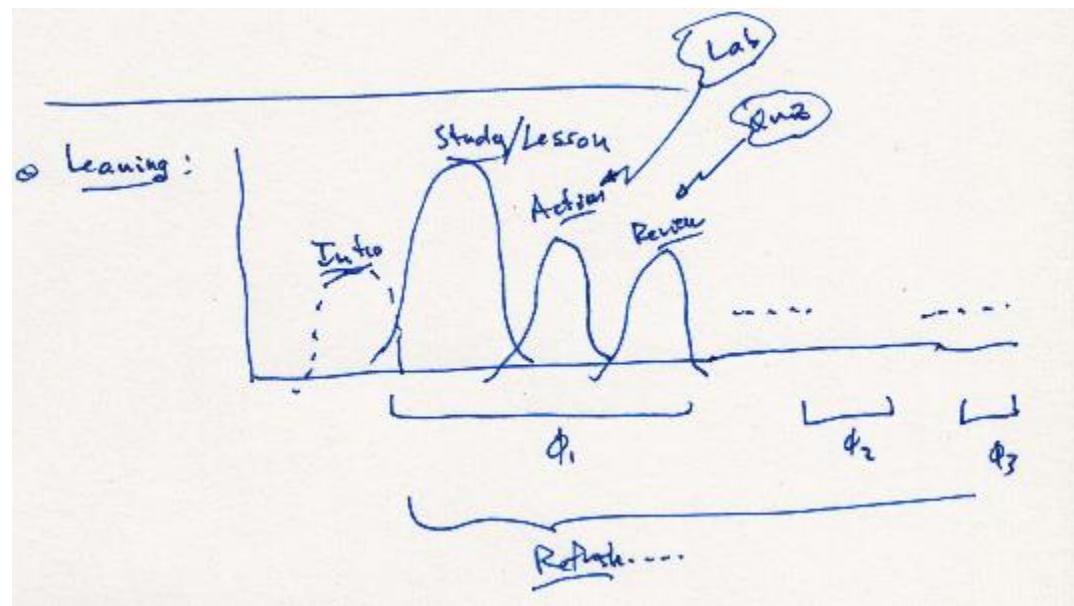
- The result is that in OO we only make the functions (behavior) visible, and encapsulate the implementation (and data).

Frameworks, Patterns, and Libraries





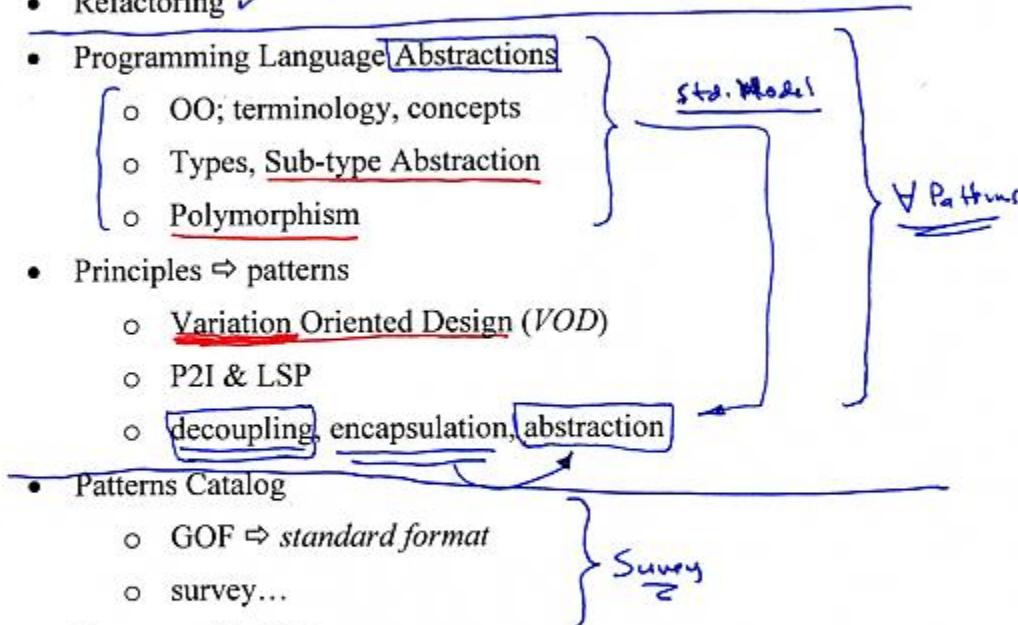
- Software Laws → Principles → Metrics



- Learning requires several repetitions, refresh the knowledge

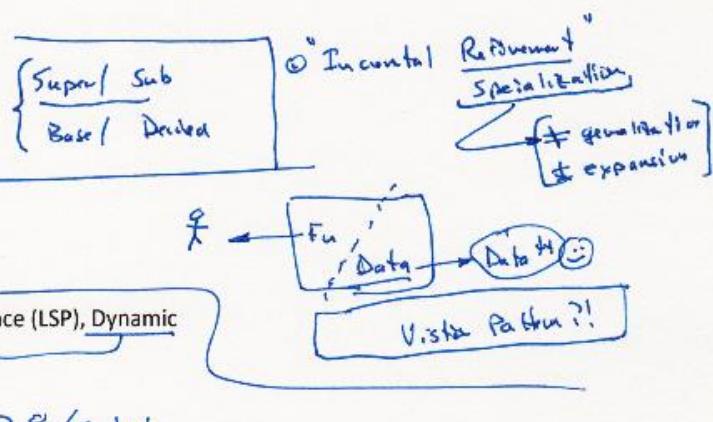
Day 2: OO programming Model and laws of Software & Design.**Day 2:**

- Quiz & Review
 - Pattern languages ✓
- Refactoring ✓
- Programming Language Abstractions
 - OO; terminology, concepts
 - Types, Sub-type Abstraction
 - Polymorphism
- Principles ⇔ patterns
 - Variation Oriented Design (VOD)
 - P2I & LSP
 - decoupling, encapsulation, abstraction
- Patterns Catalog
 - GOF ⇔ standard format
 - survey...
- Resources for Patterns
- Reading (GOF)
 - Homework



OO Model: three requirements

- Types & sub-types
 - LSP → "isA"
- Encapsulation
 - (Fn + Data)
 - Behavioral model
- * Polymorphism: based on inheritance (LSP), Dynamic
 - Msg::Method - Binding
 - \equiv abstraction (1:n)
 - Interface :: What $\xrightarrow{\text{!}} / \underline{\text{Contract}}$
 - Method :: How



• The OO Programming Model

Binding:

- Fundamental aspect of programming
 - Abstraction :: Concrete realization
 - Name :: meaning
- Examples:
 - "C" → Static binding |_{CT}
 - (1:1) name :: Function
 - Name \equiv function (!)
 - Overloading → static binding |_{CT}
 - "OO" → dynamic binding |_{RT}
 - Polymorphism
 - (1:n)
 - N – from LSP
 - Name (msg) :: Method (implementation)
 - Fixed (interface) :: changing (impl.)
 - ≈ Open/closed

Times...

Pre-processor
Compile
Load
Link
Run - Time

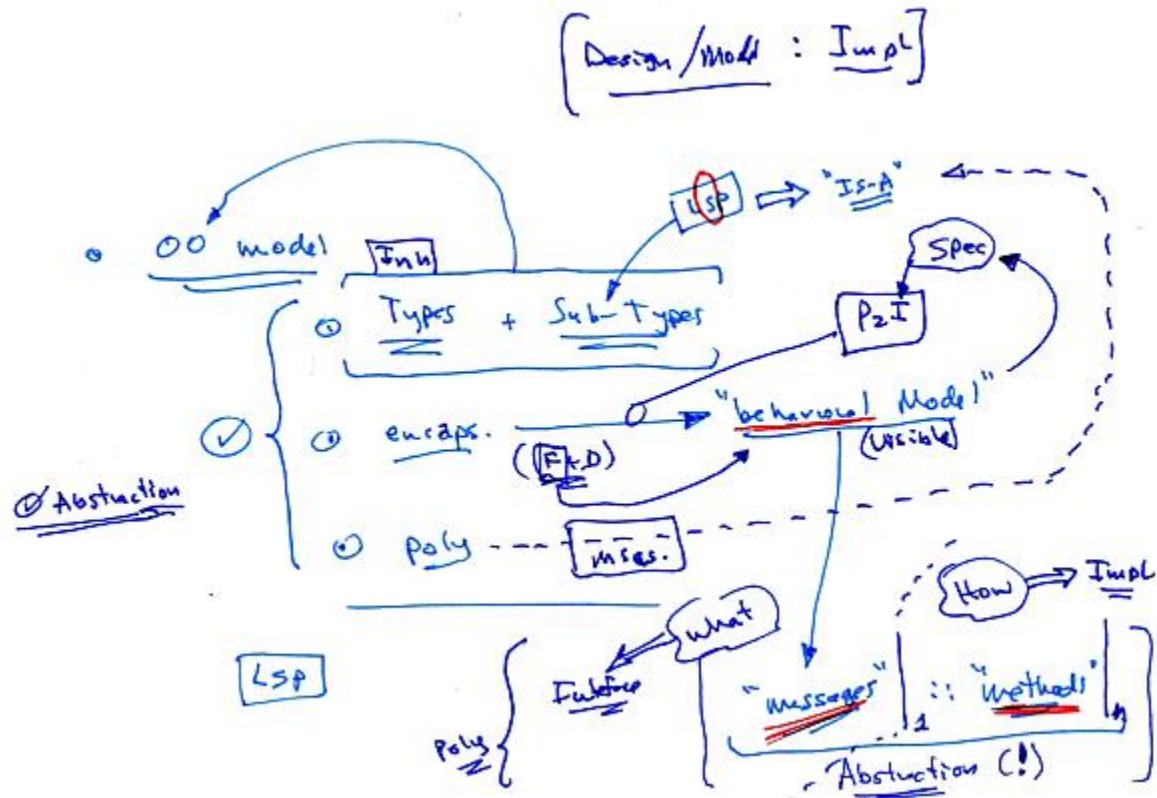
• Abstraction (✓)
(1:n)

S

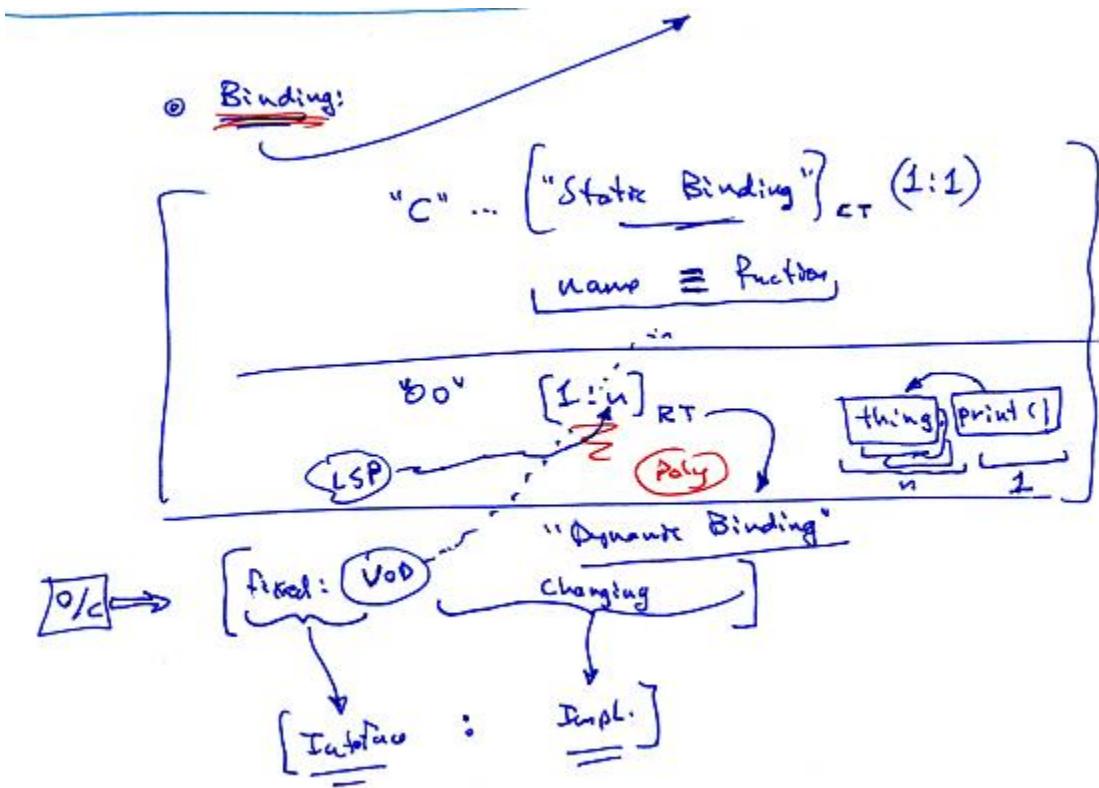
Fixed (interface) :: changing (impl.)

≈ Open/closed

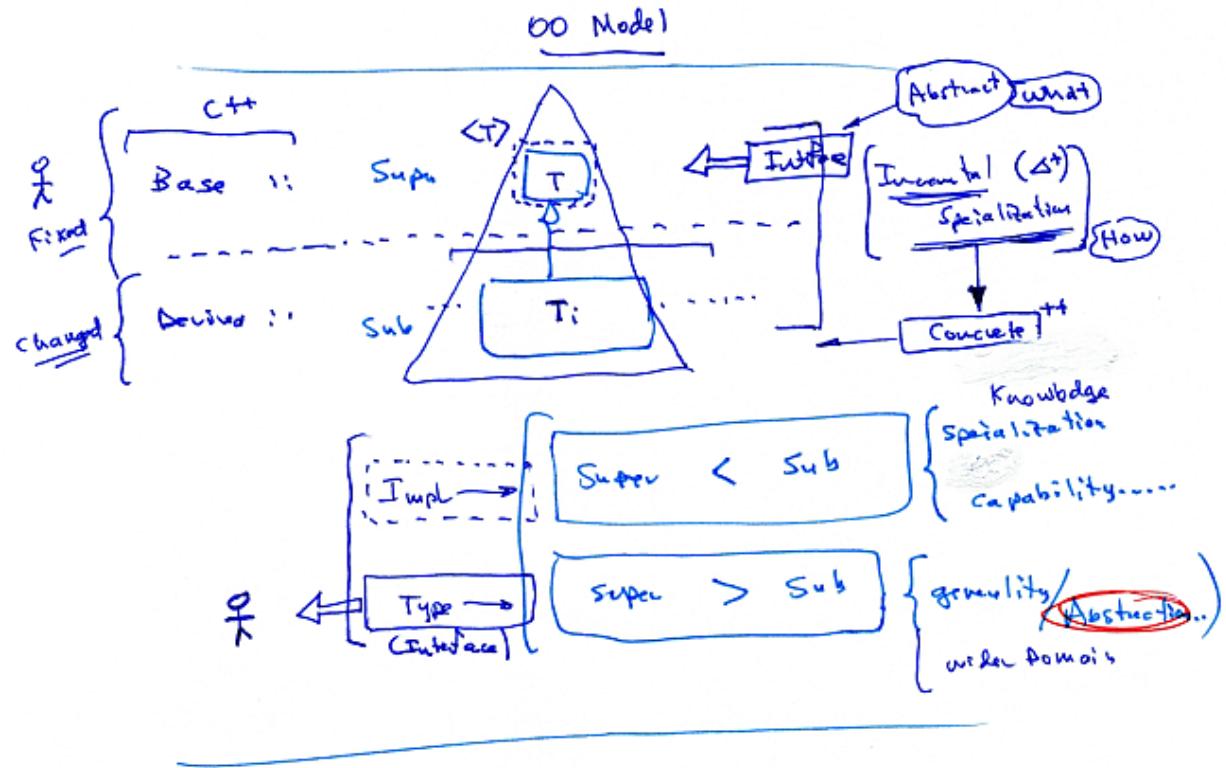
• Binding



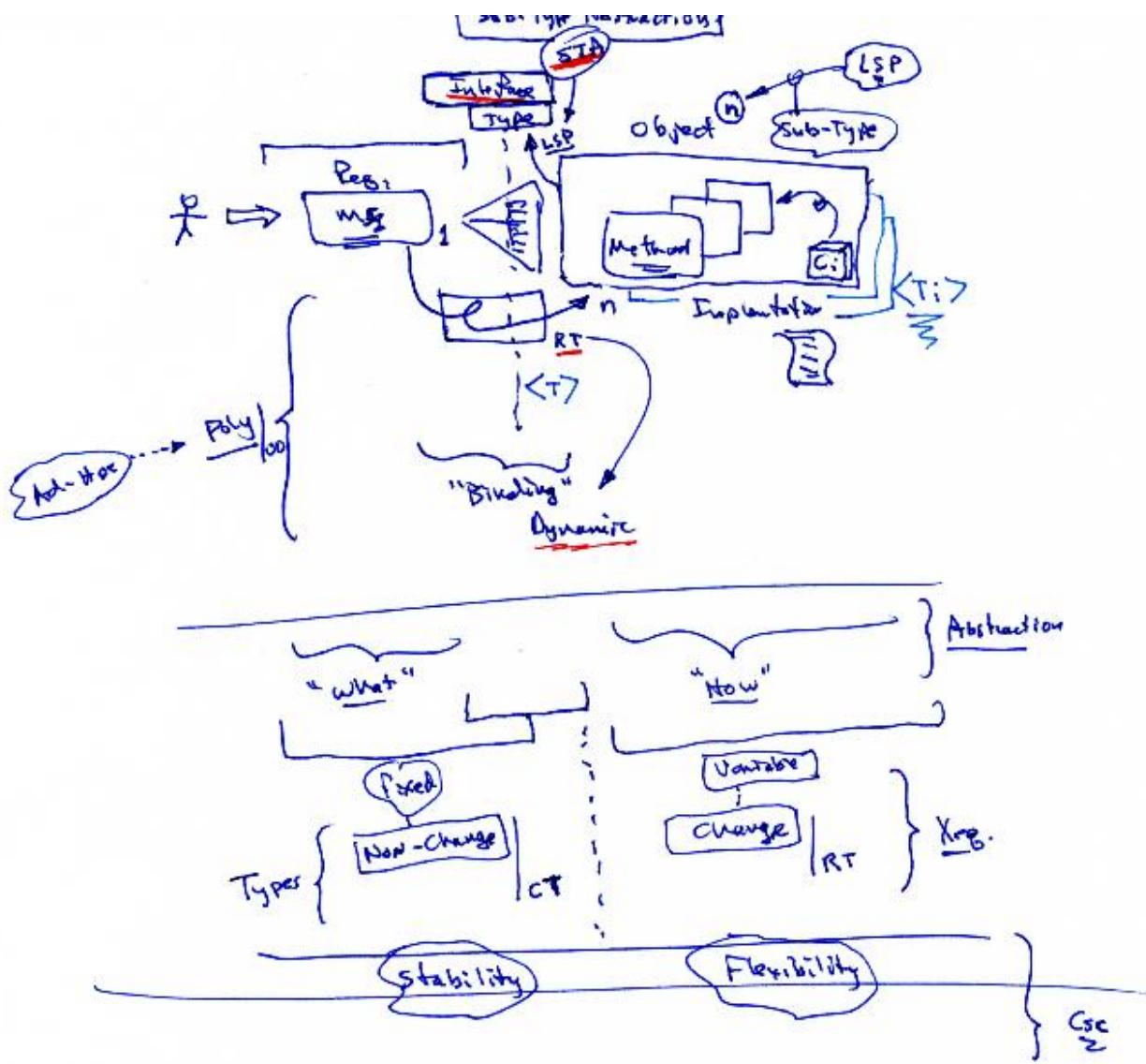
- Definitions and features of OO software



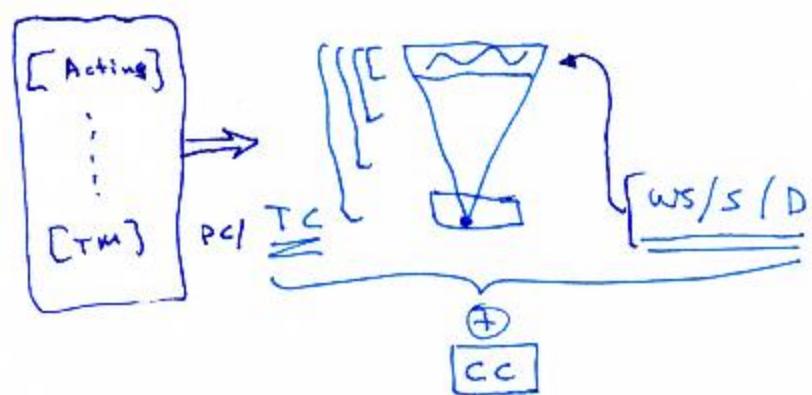
- Dynamic Binding



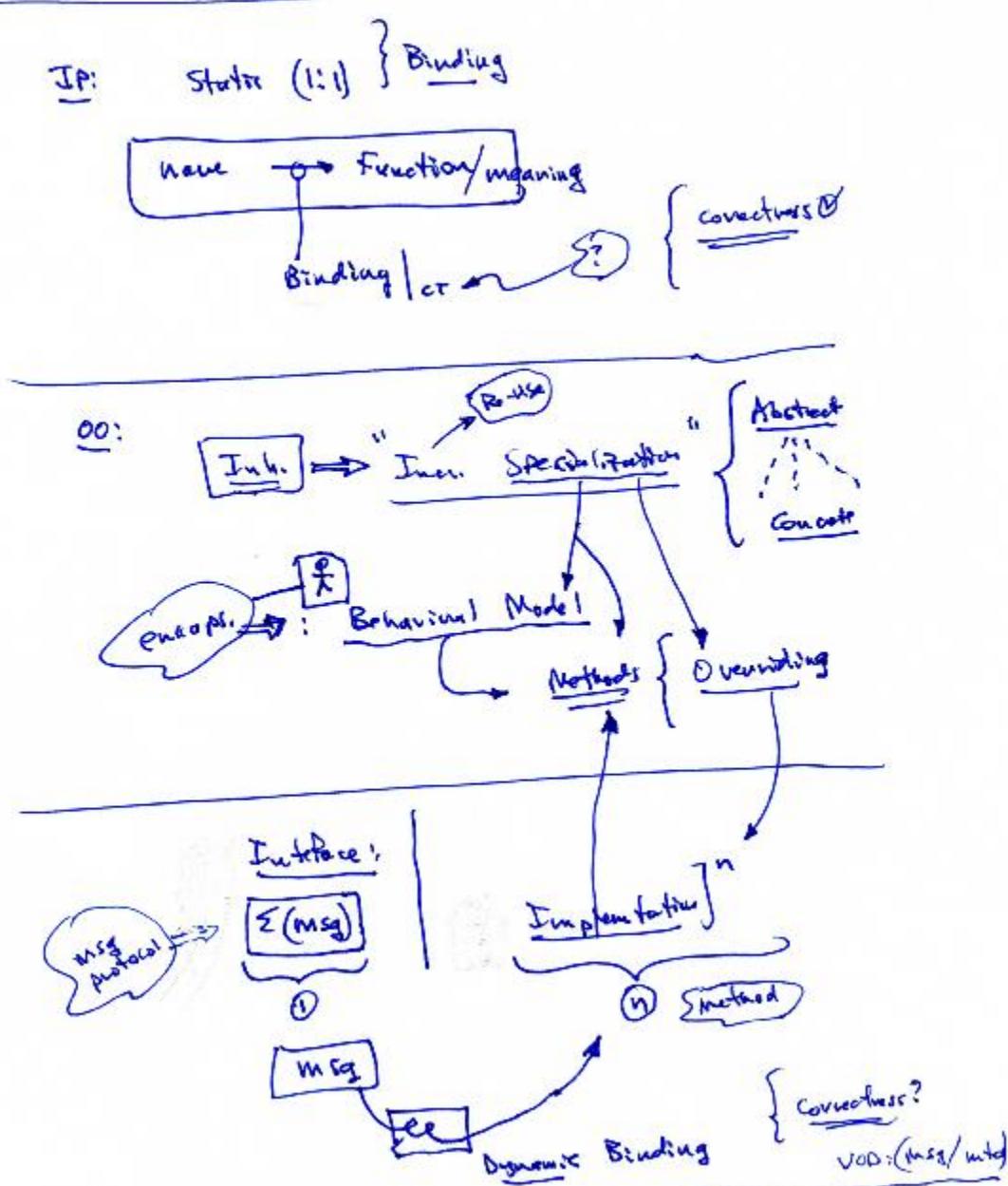
- Class Hierarchies



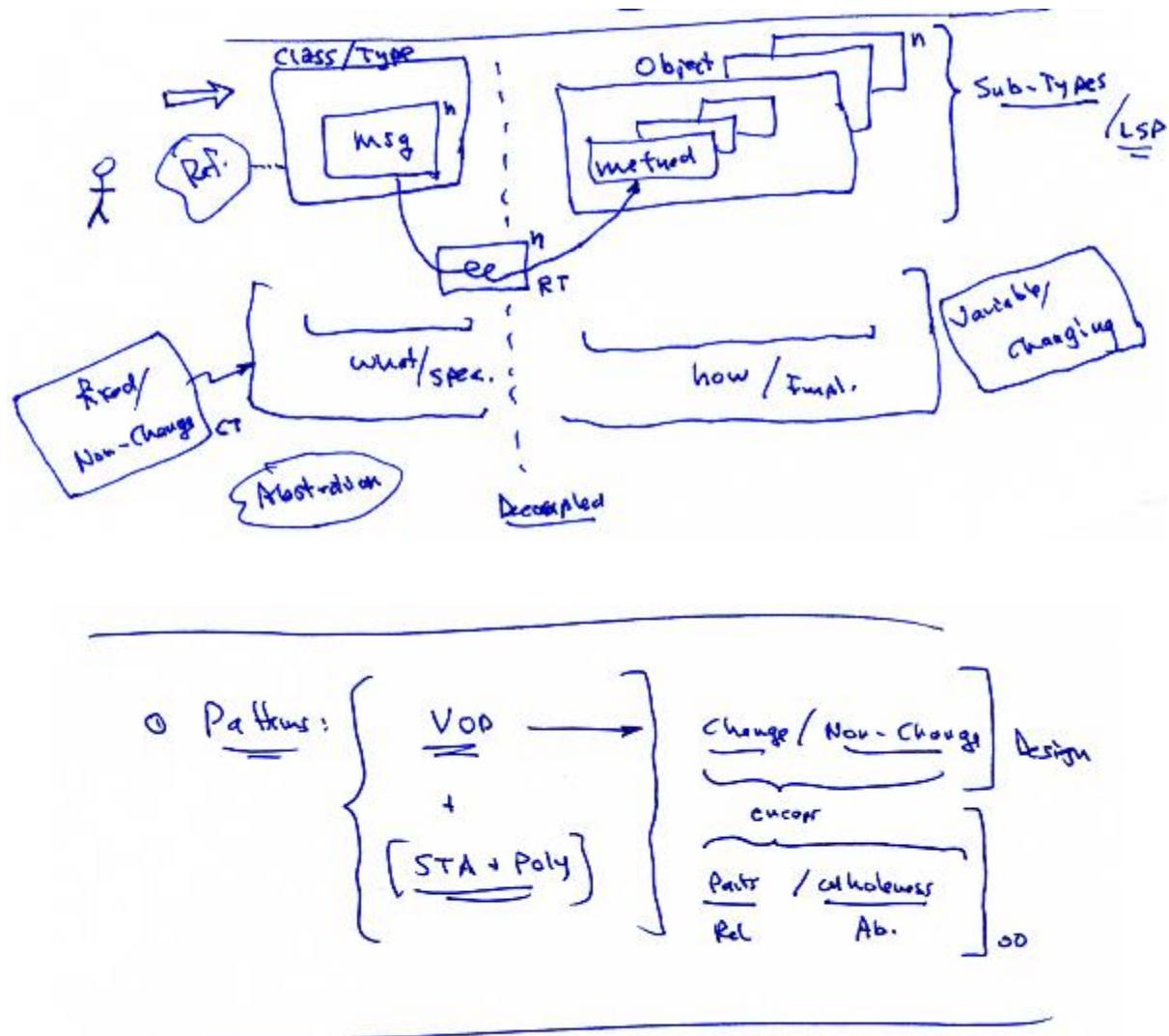
- Sub-type abstraction (STA)



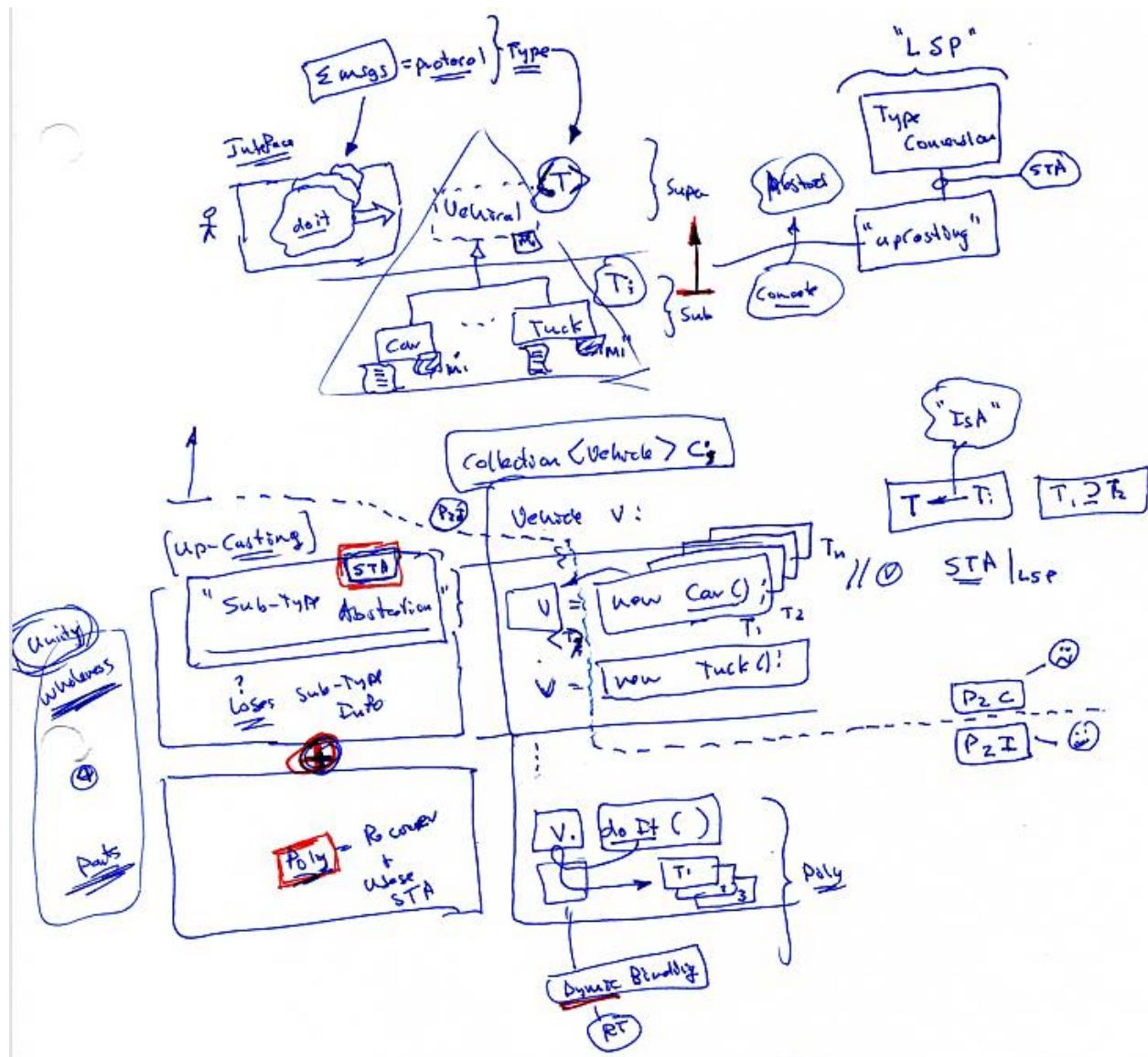
- SCI relationship to STA



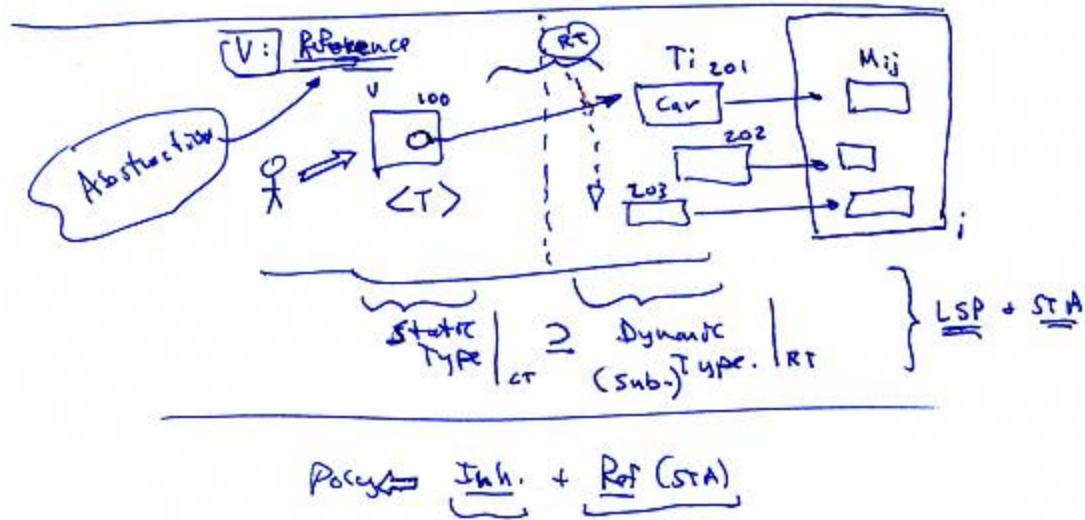
- Dynamic Binding



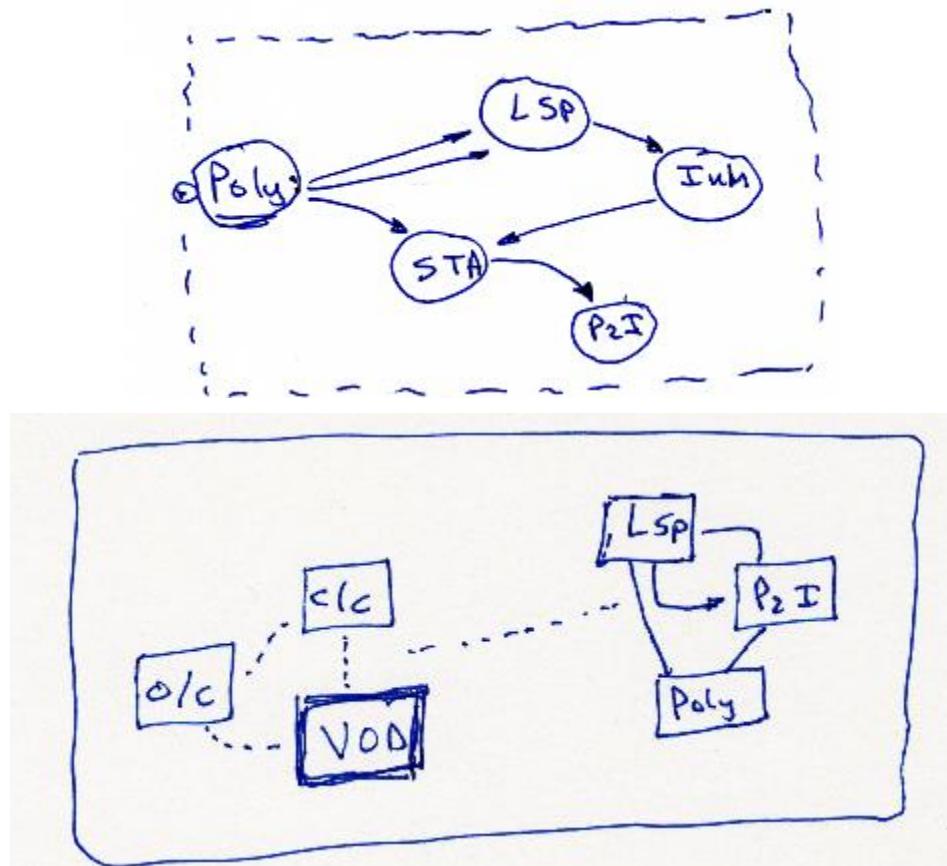
- The common basis of all patterns
- VOD = Change + non-change



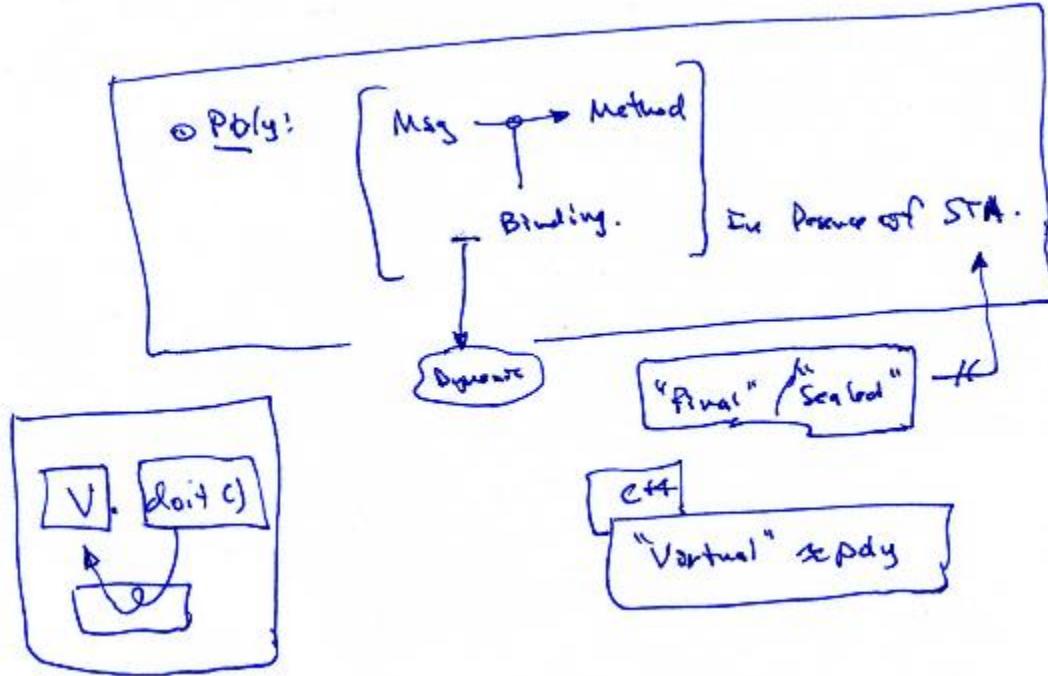
- STA + Poly \Leftrightarrow Wholeness + Parts
- Example of STA
- STA = *upcast*



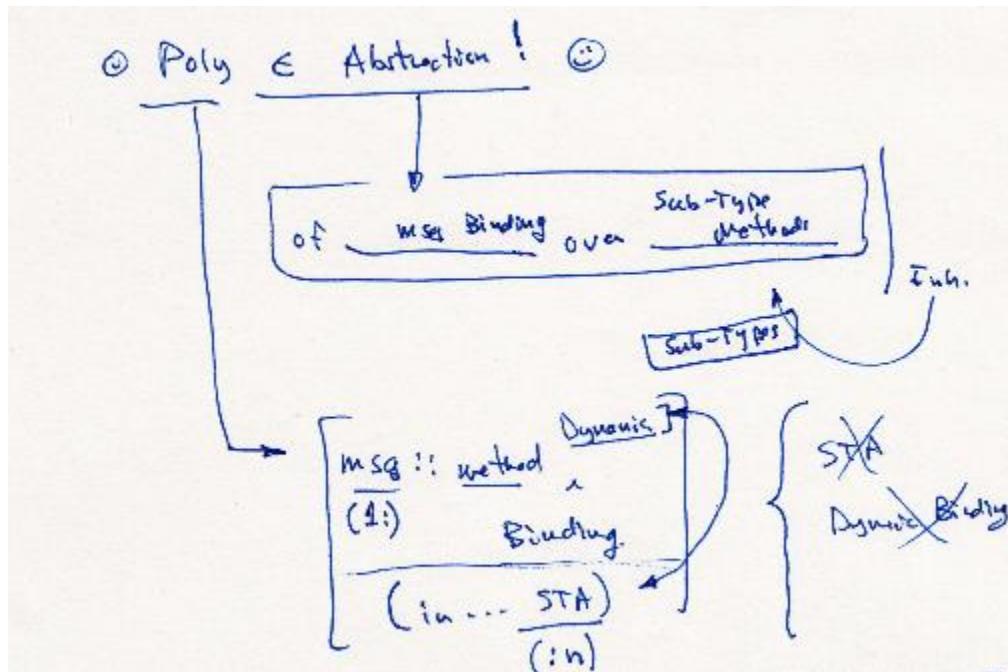
- References are an abstraction over values
- Poly & STA always/only occur together



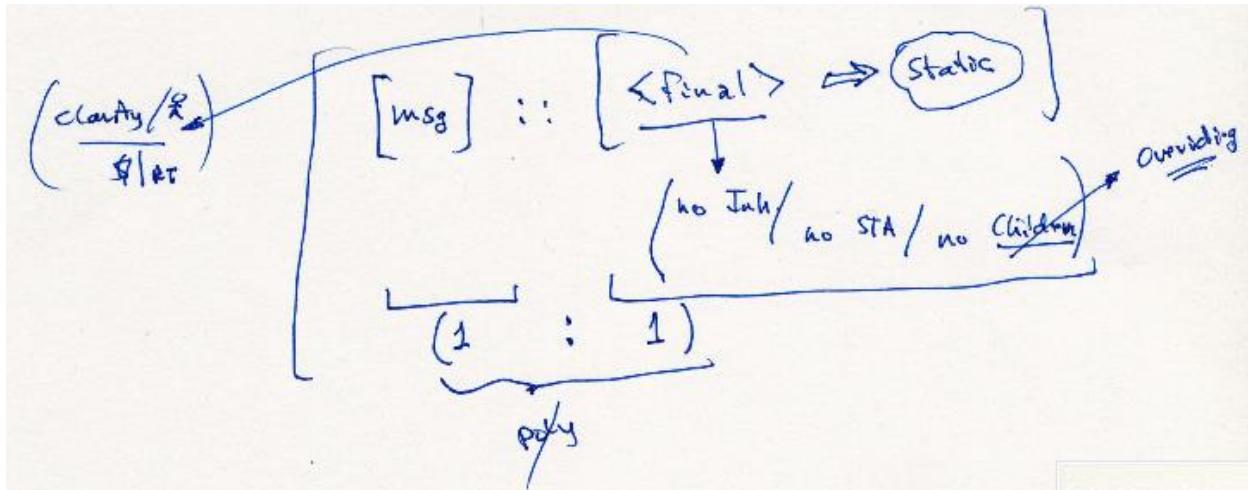
- Cluster of related concepts



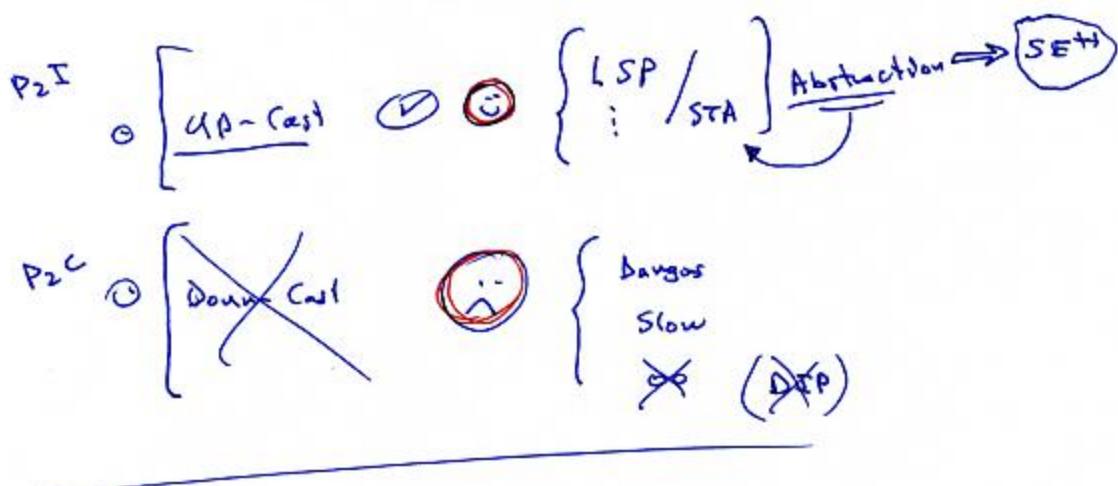
- Technical definition of polymorphism



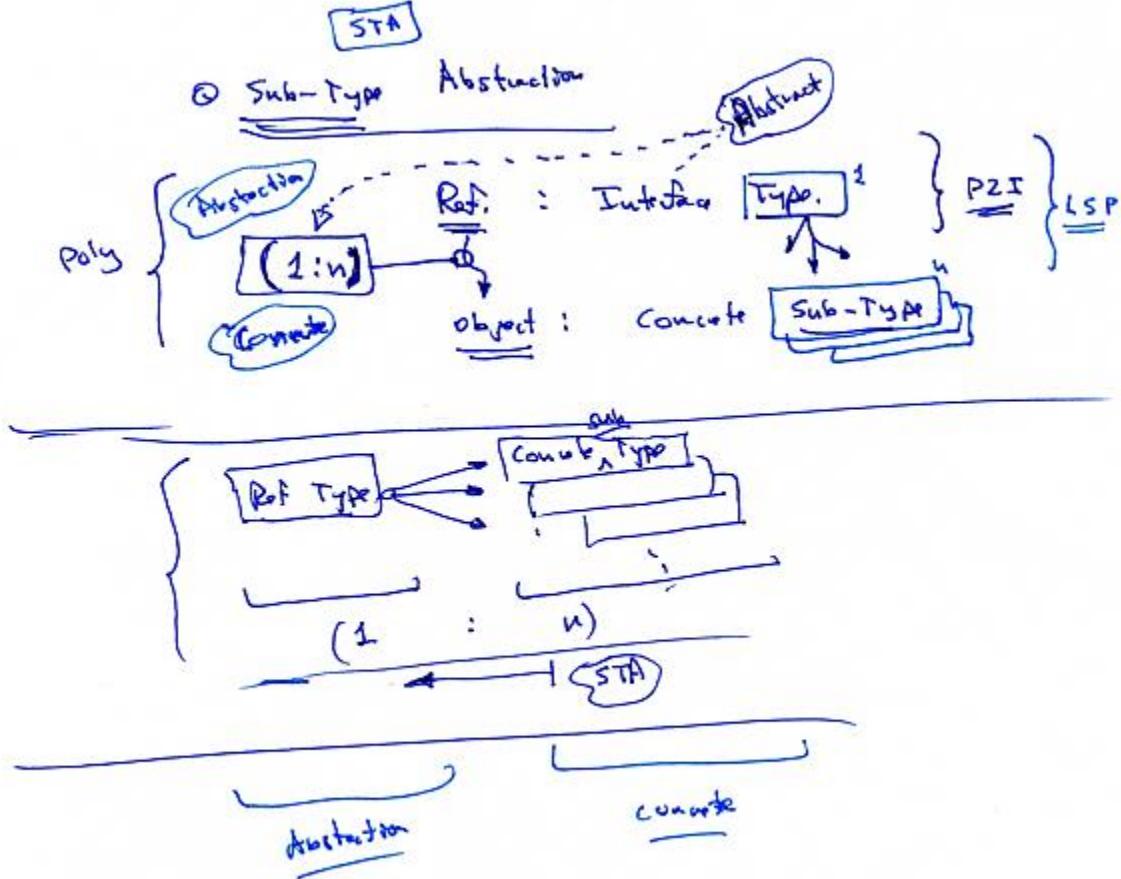
- Polymorphism is a (fundamental) abstraction



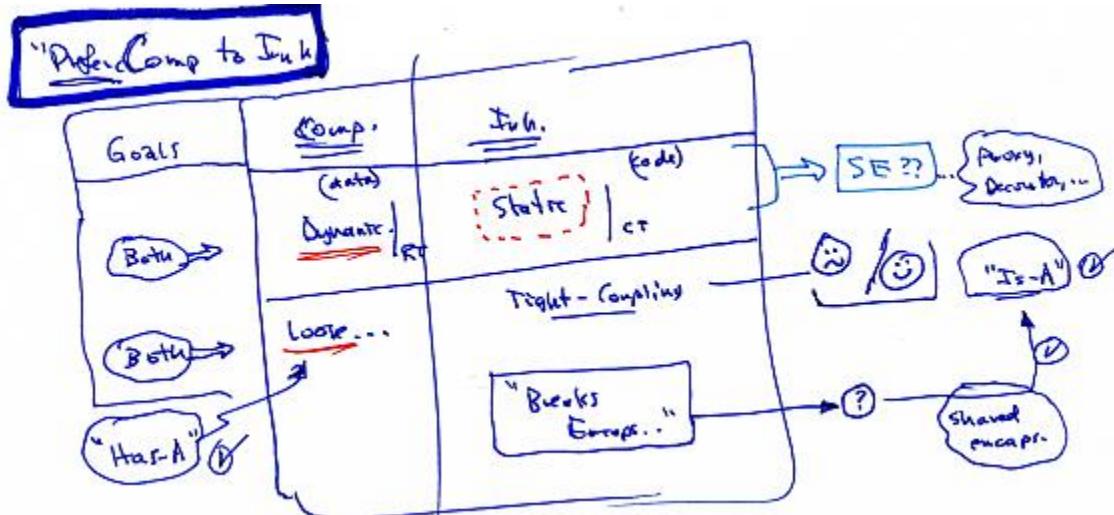
- Static binding (<final>) is not an abstraction



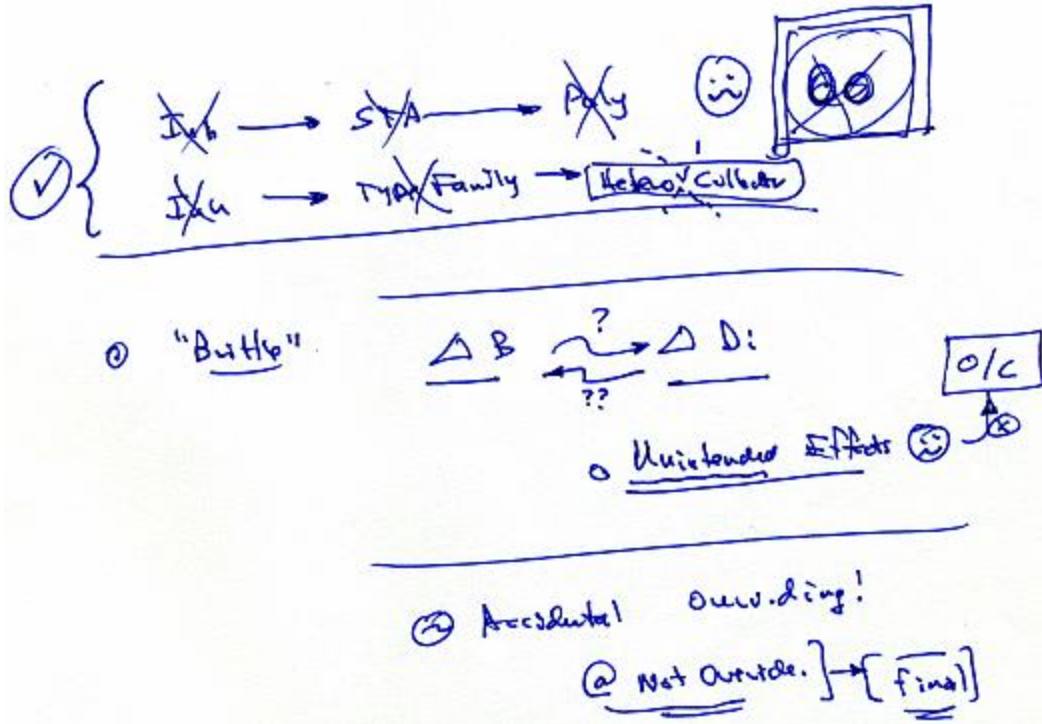
- LSP + STA = up-casting = Good, down-casting ... not good



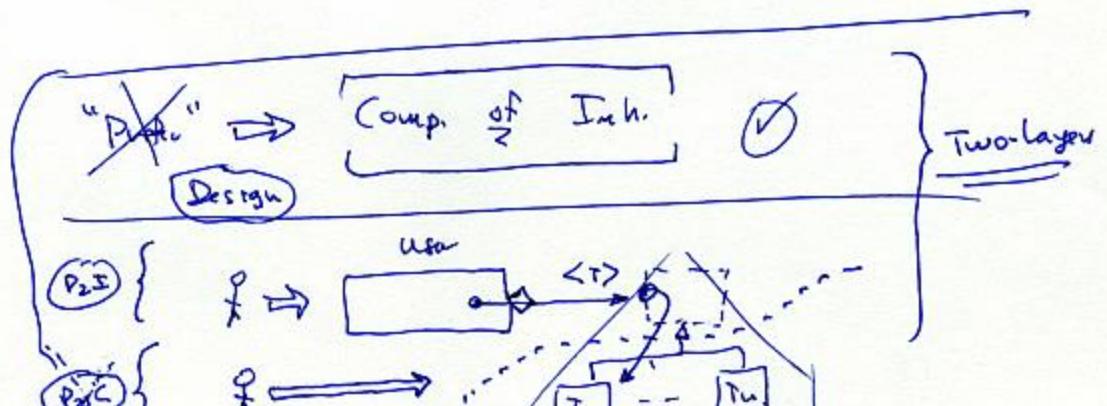
- P2I & STA



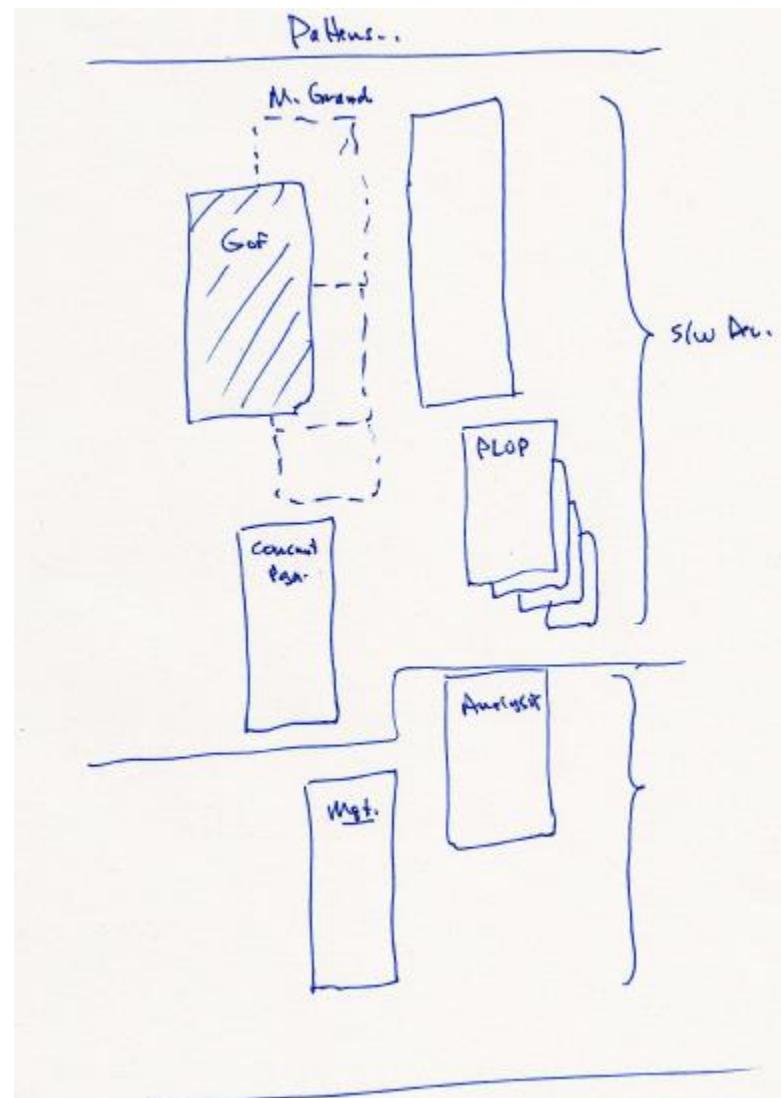
- GOF: Prefer Composition to Inheritance
- ⇒ Composition of Inheritance



- Without inheritance; no LSP \rightarrow no STA \rightarrow no Poly \Rightarrow no OO!
- There are issues with inheritance – *Brittle* class relationships



- Both composition & Inheritance (*Has-A*, *Is-A*) are important design concepts, and both should be supported and used in design.



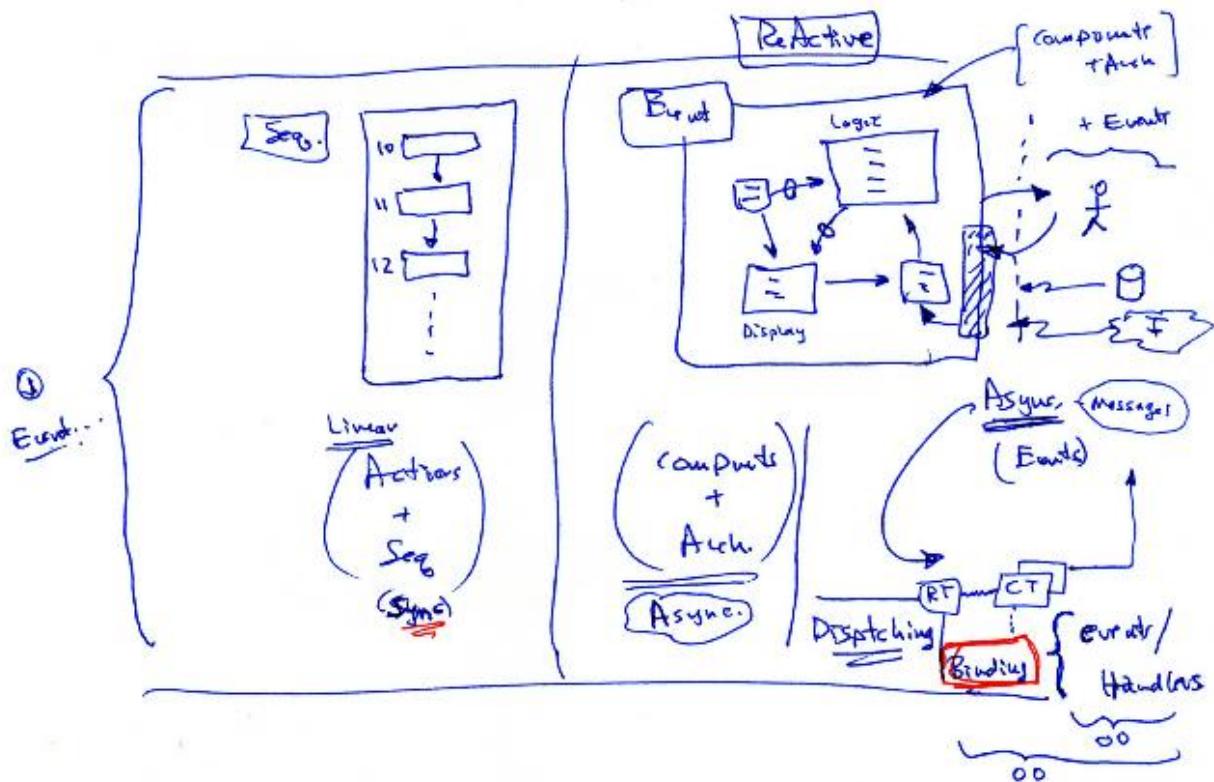
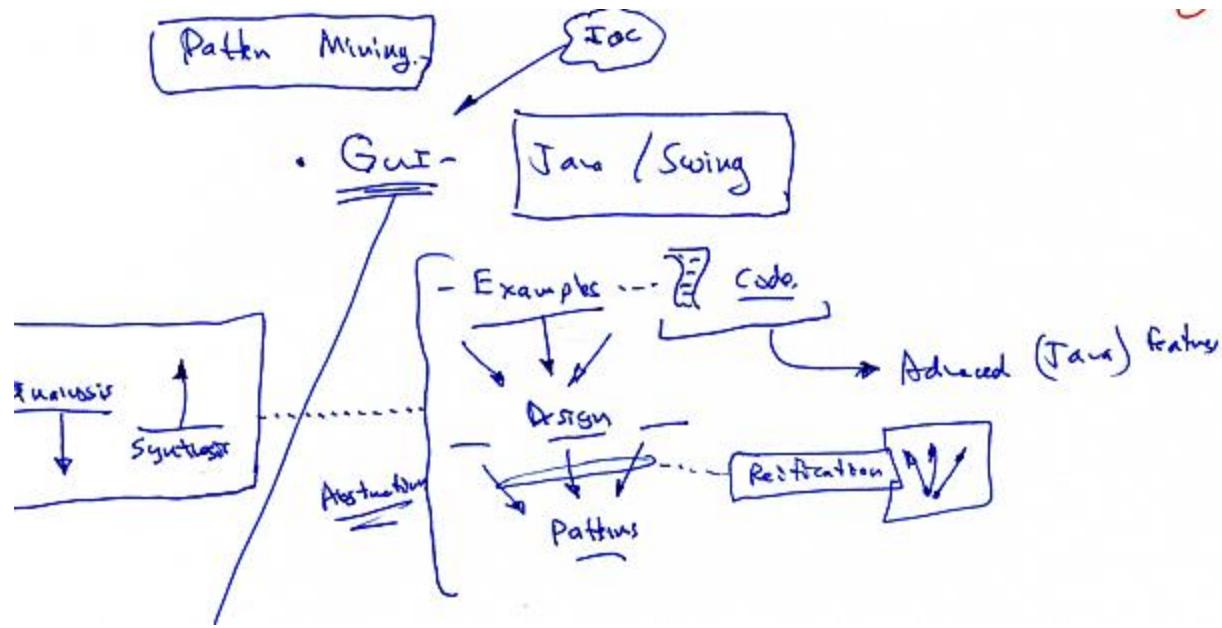
- GOF is the main reference for patterns, but there are also many other sources.

Day 3: Pattern Mining & Swing, Observer Pattern

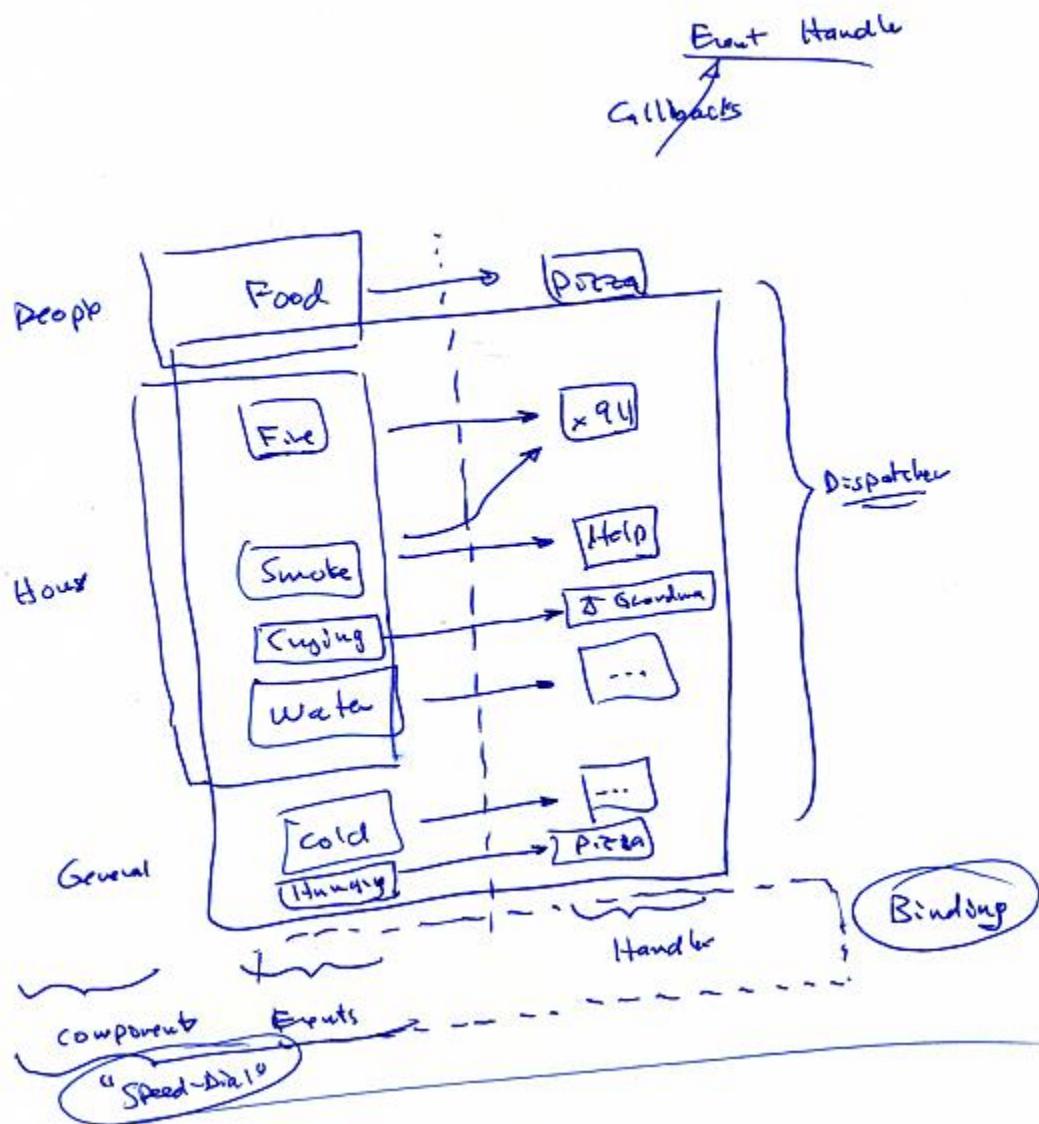
Daily Topics

Day 3: Pattern Mining & the Observer Pattern

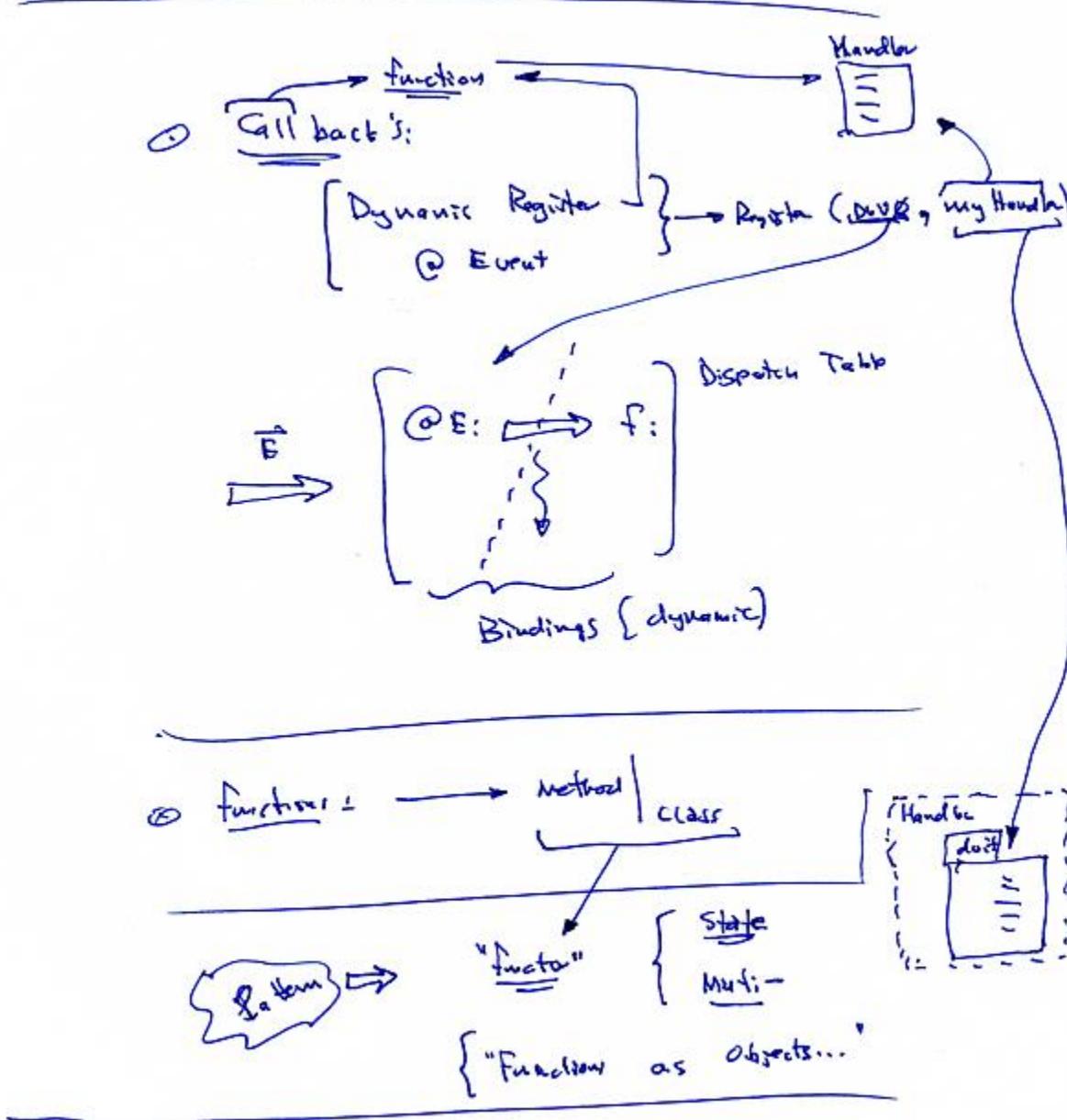
- Quiz & Review
- Pattern Mining
 - Swing
- Event driven Programming
 - event architecture
 - functor pattern
 - Java event model
 - *ActionListeners*
 - Button examples
 - = *observer pattern*
 - Java idioms for listeners → OO
- Observer Pattern \Rightarrow GOF
 - Standard pattern template
- Patterns in Swing:
 - MVC
 - command
 - observer
 - Composite
 - Strategy, Bridge, ...



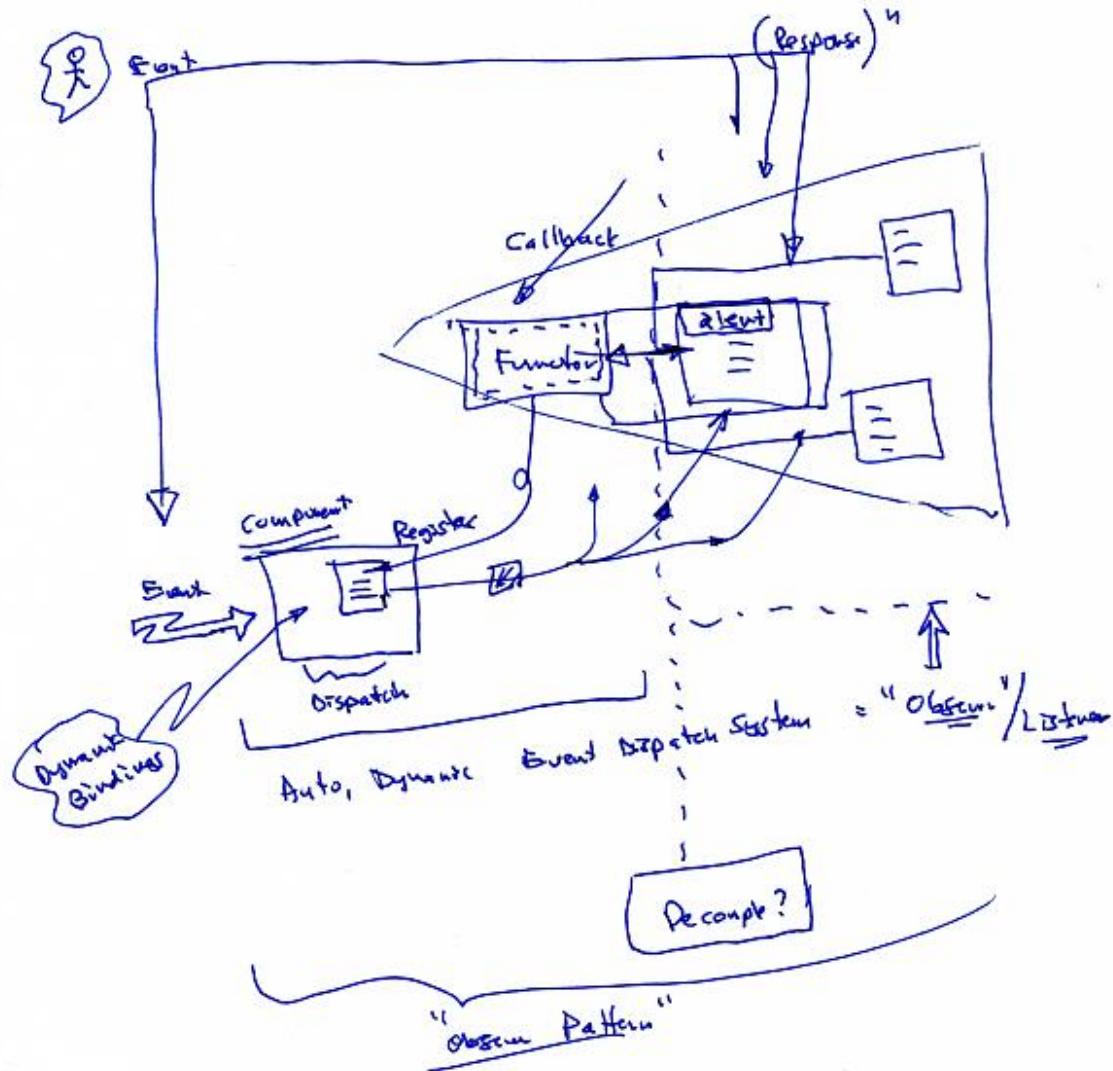
- Event driven programming model



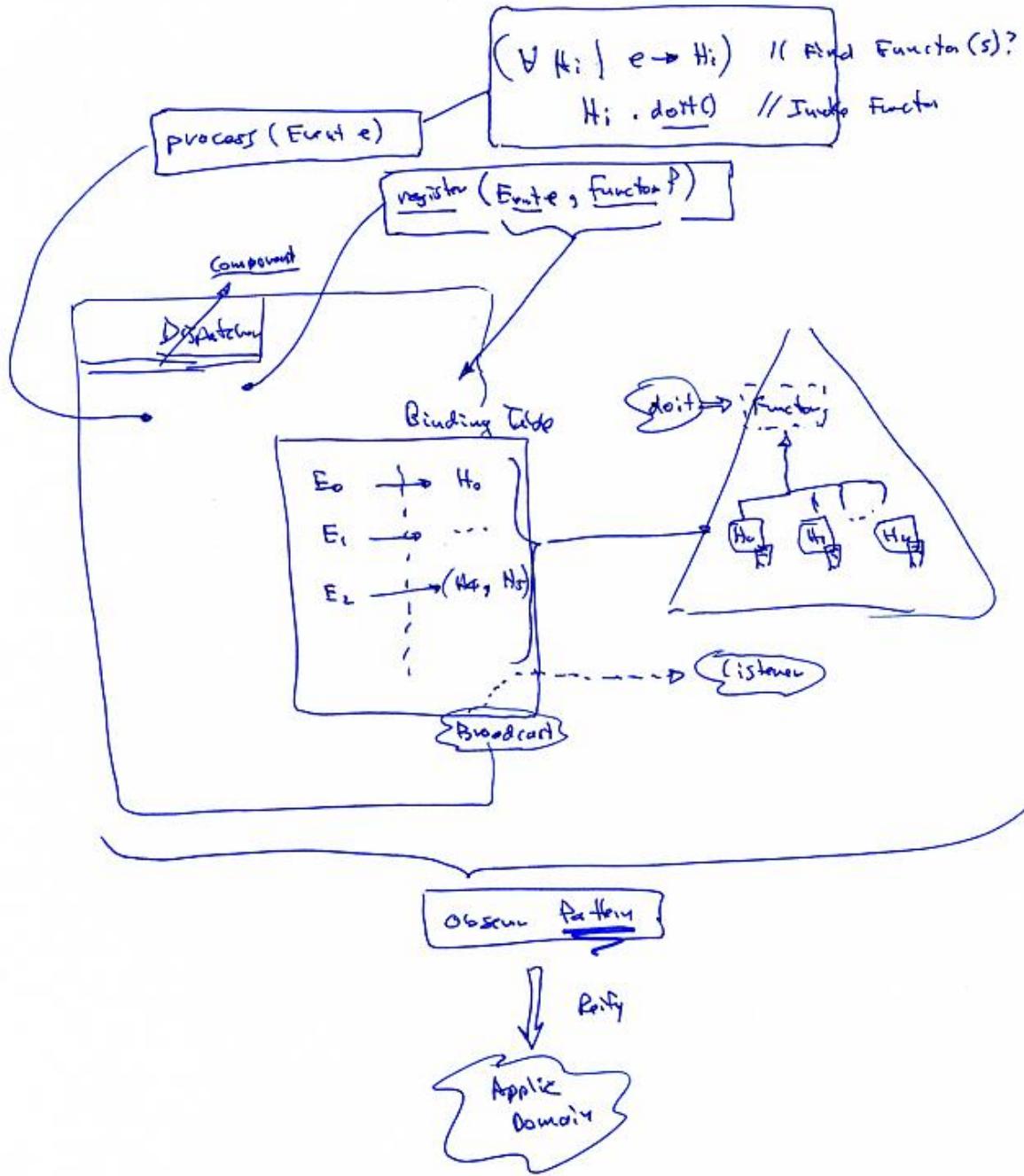
- Callbacks and event handlers



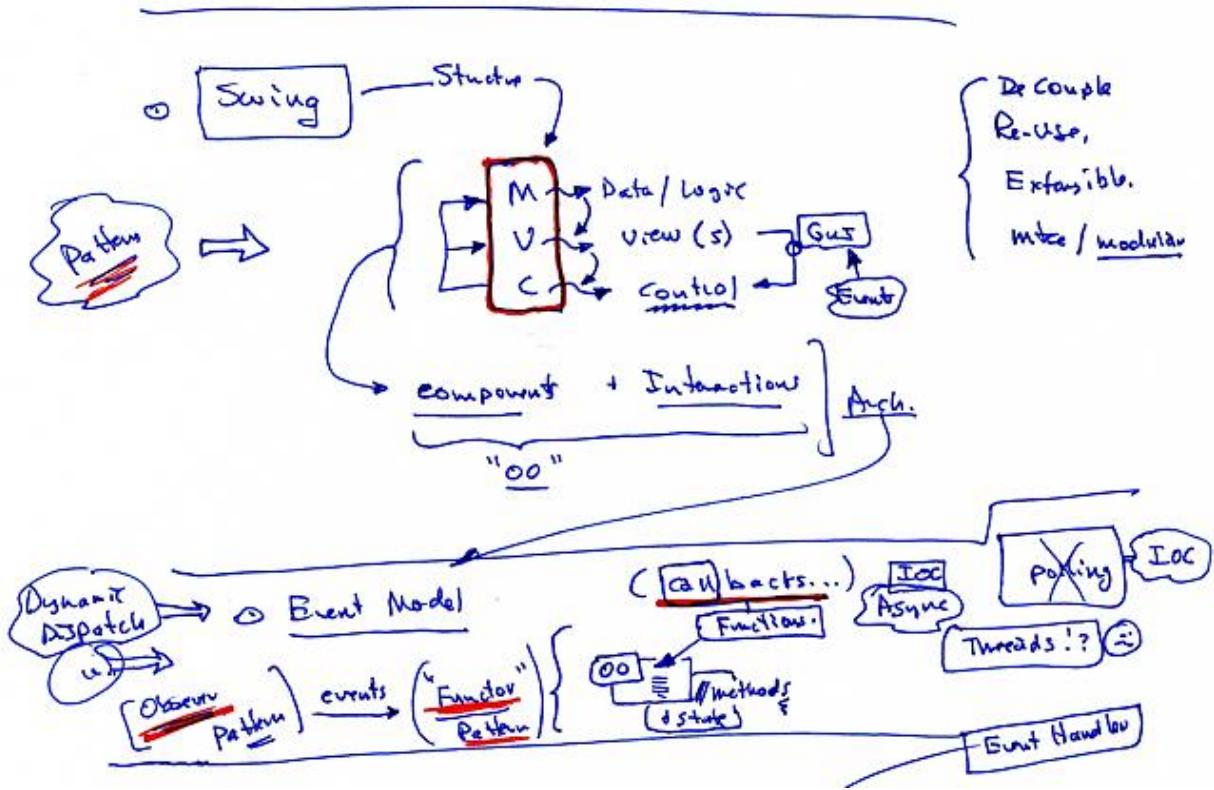
- Callbacks as event handlers
- *Callbacks* = functions \Rightarrow *functors* in OO



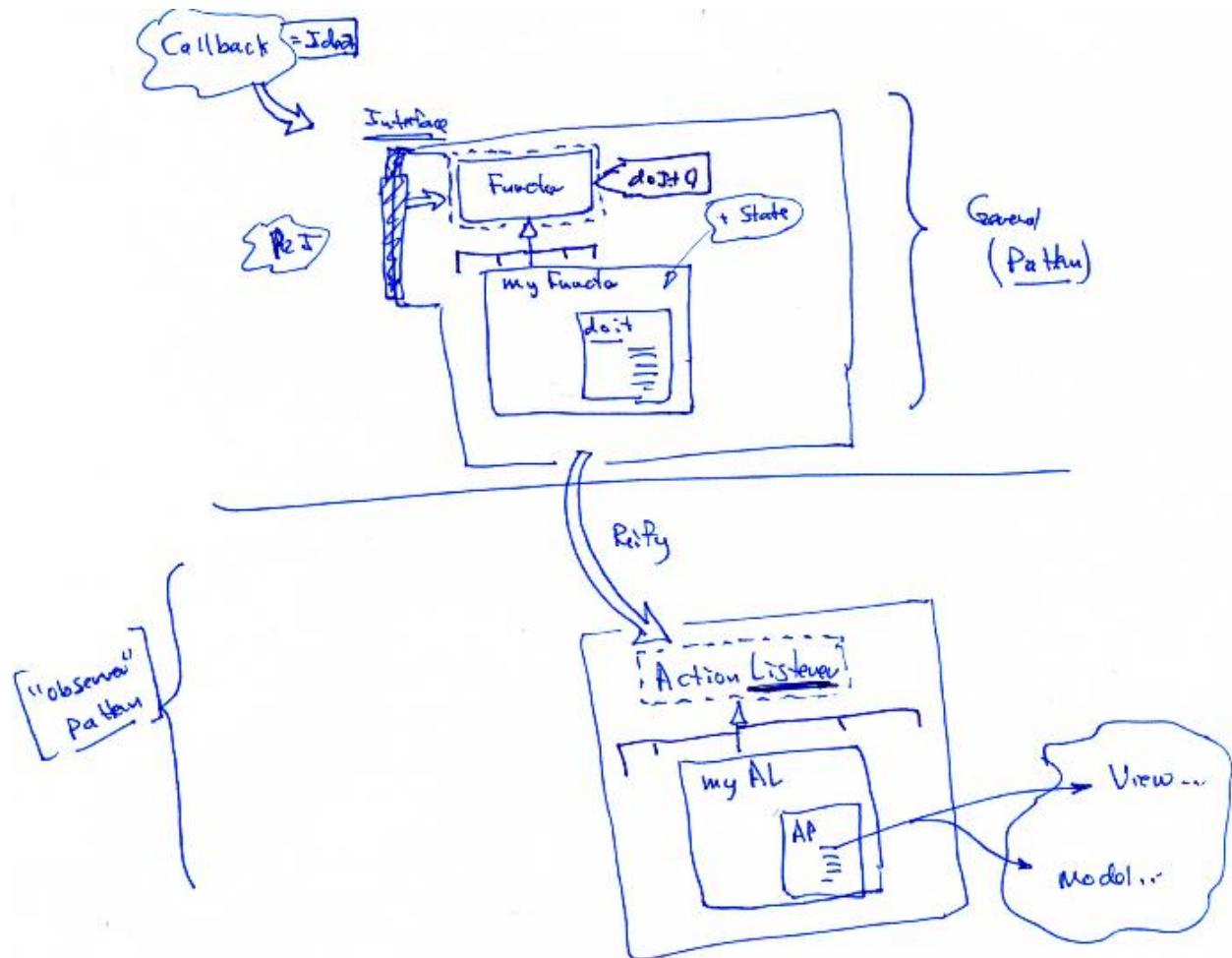
- Using a registry of functors (event handlers) for event dispatching



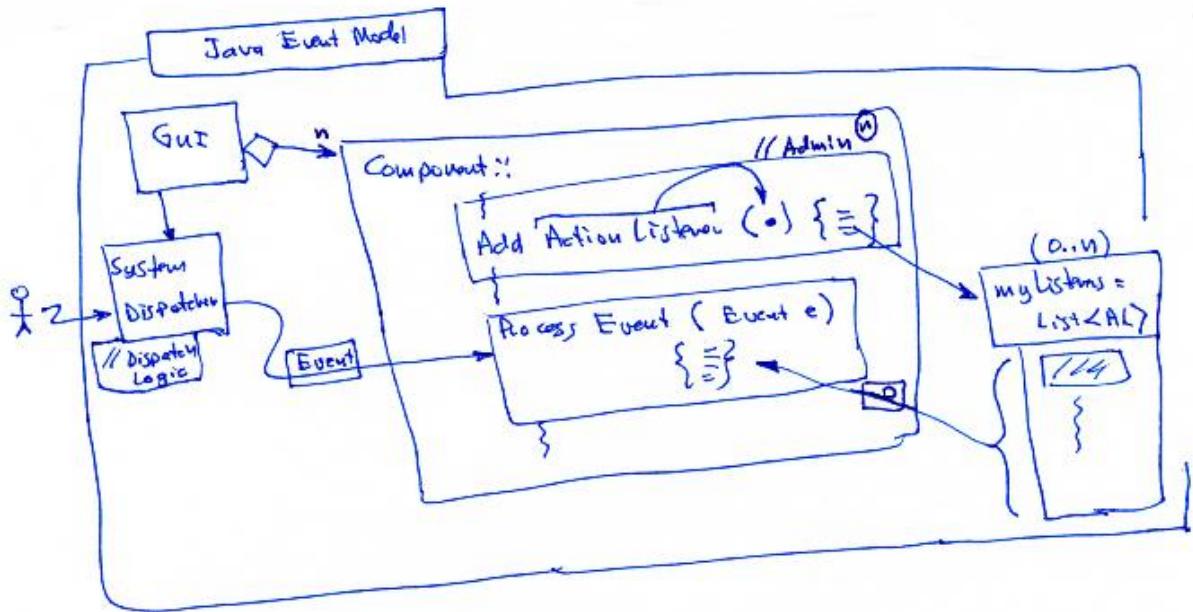
- \Rightarrow Observers (Listeners)
- + management and registration \Rightarrow *Observer Pattern*



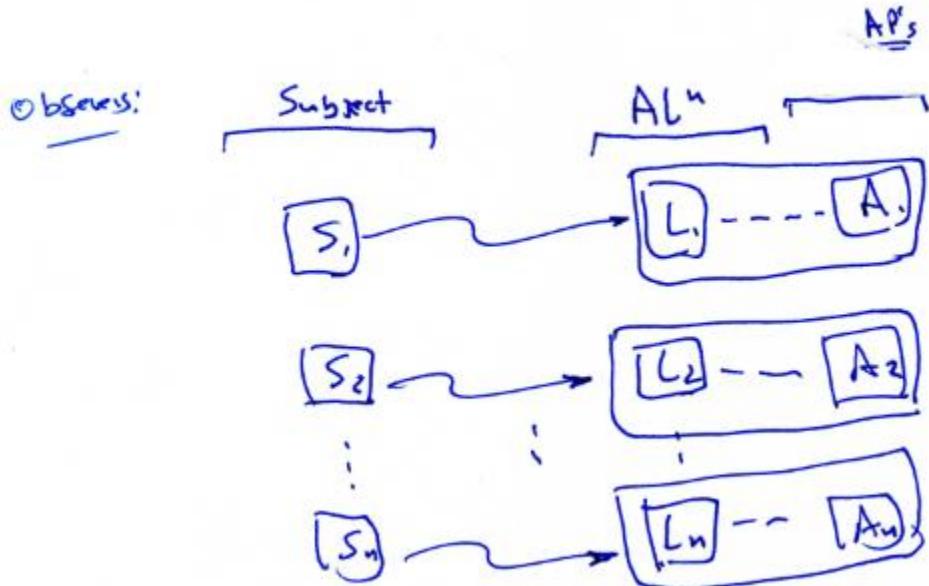
- Swing uses MVC and the Observer pattern



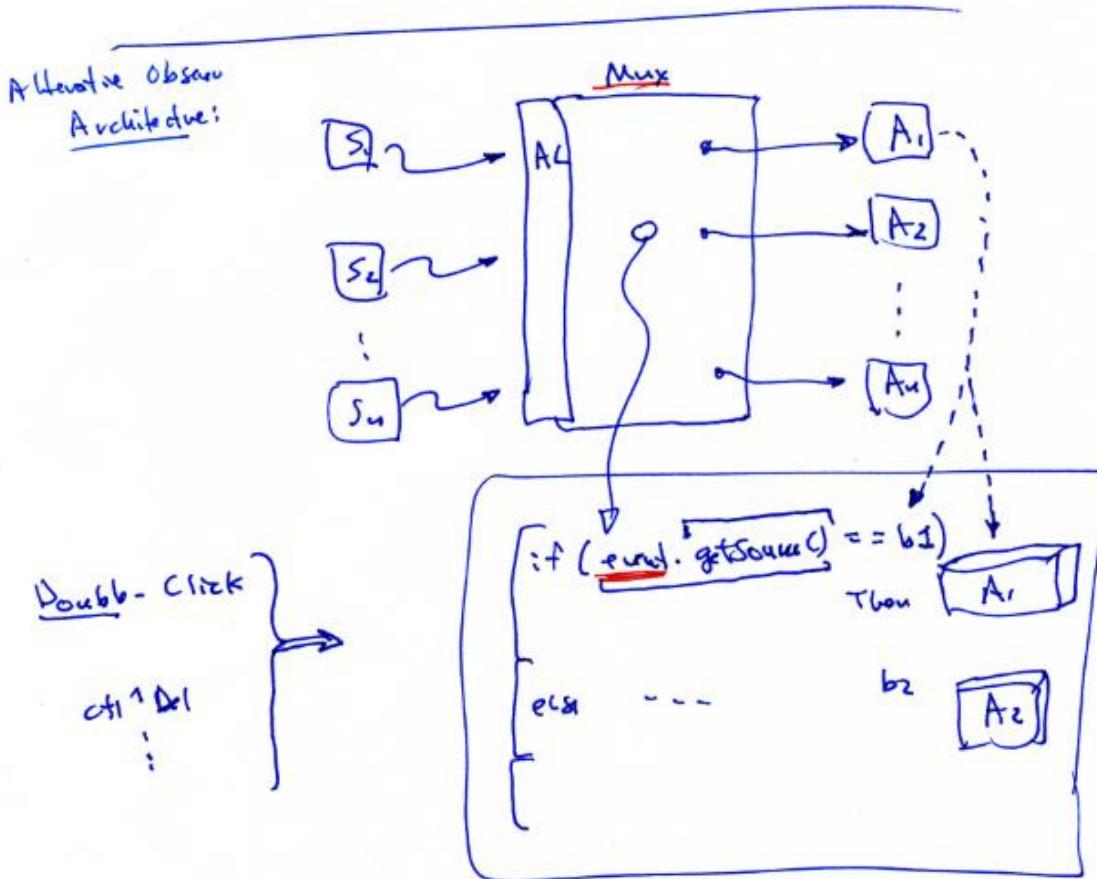
- Reify the General Observer Pattern → Swing ActionListeners



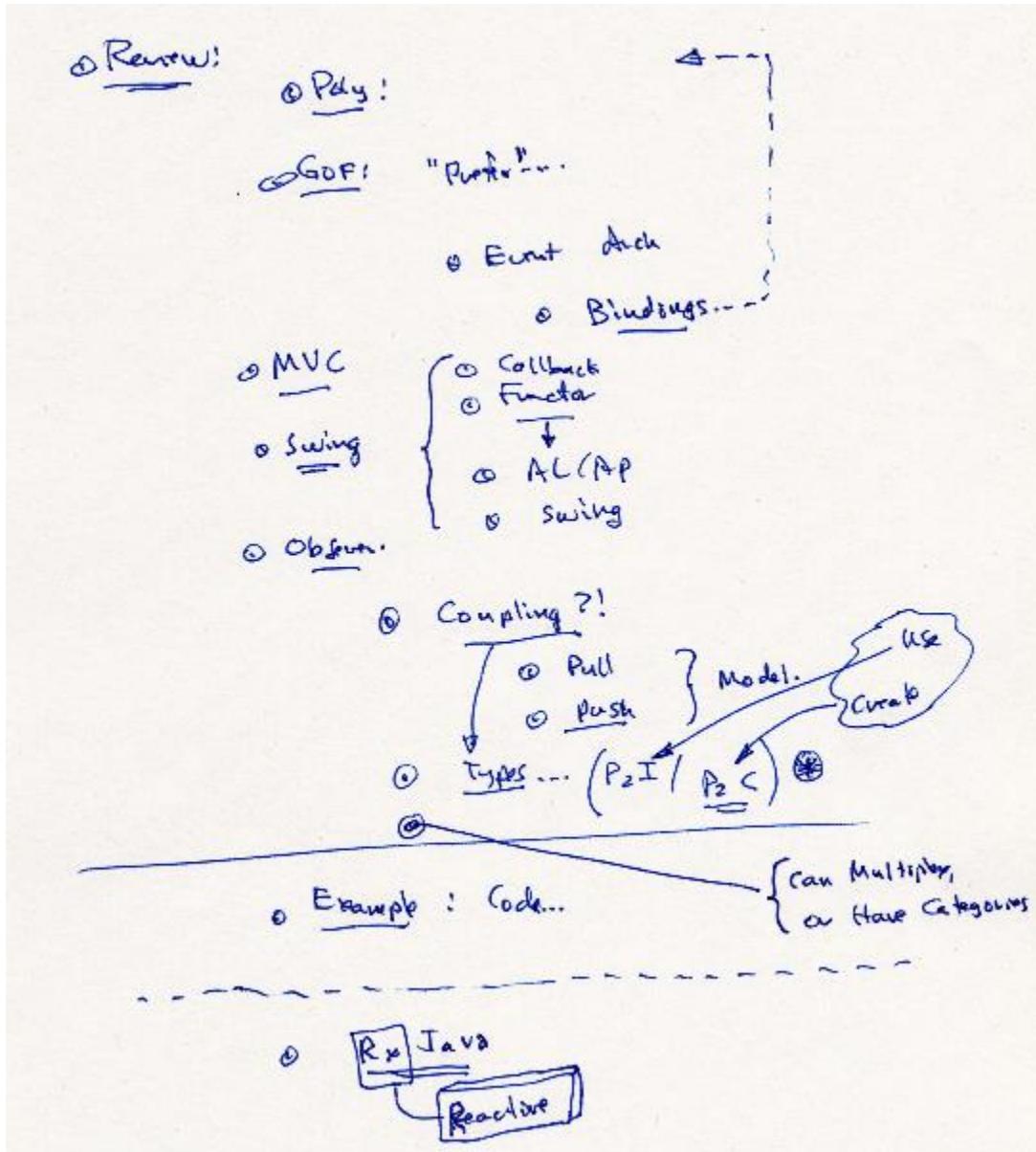
- Java Swing event model



- Individual AL's @ event (subject) = de-multiplexed (like our class & GOF)

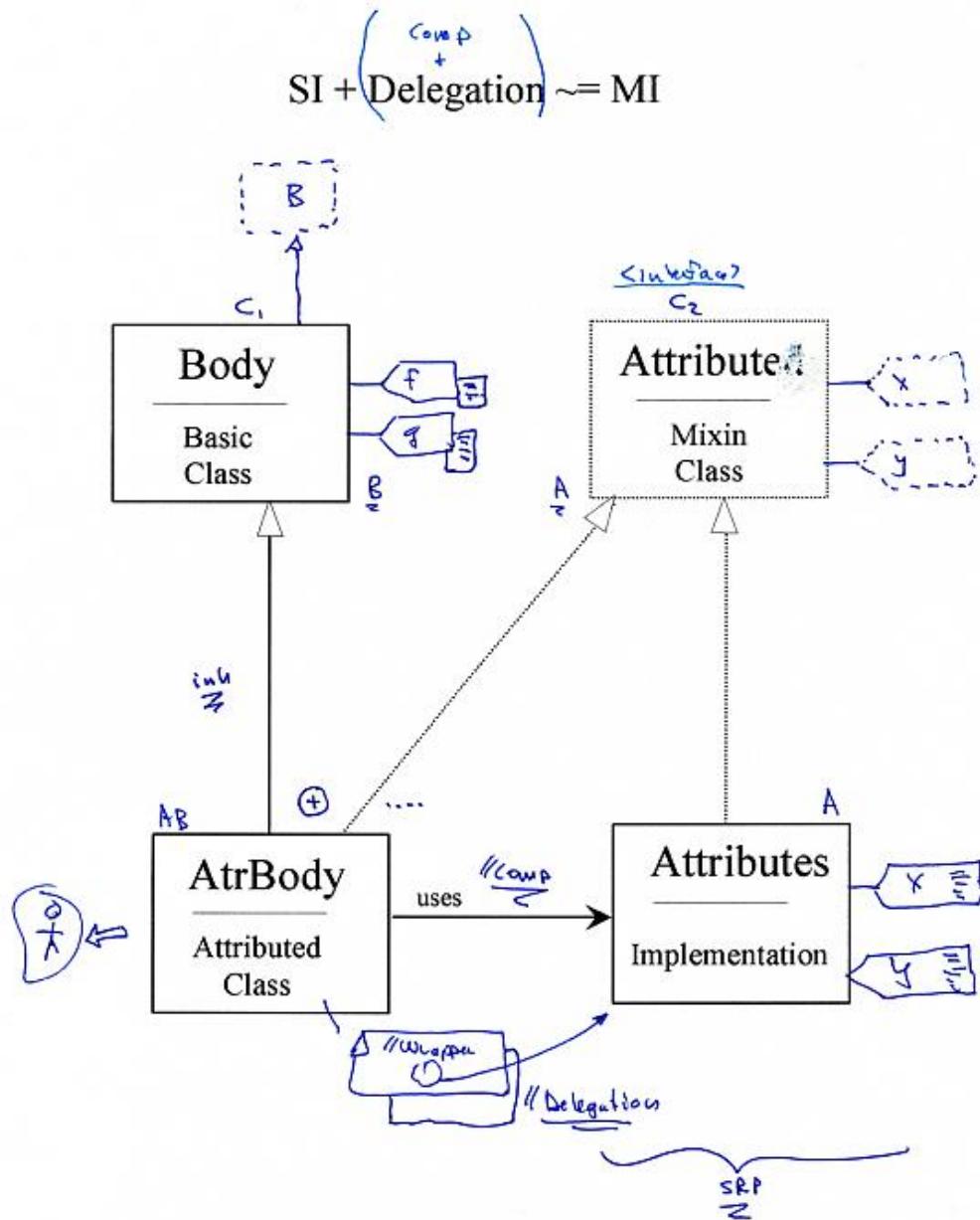


- Multiplexed Events (Event Manager), like the lab software



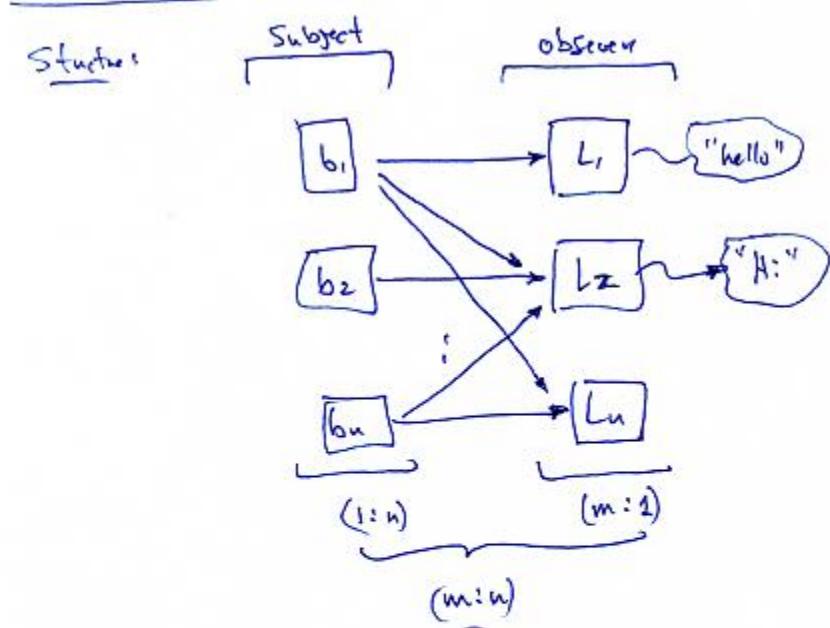
- Review of topics and development logic.

The *Fake-MI* pattern



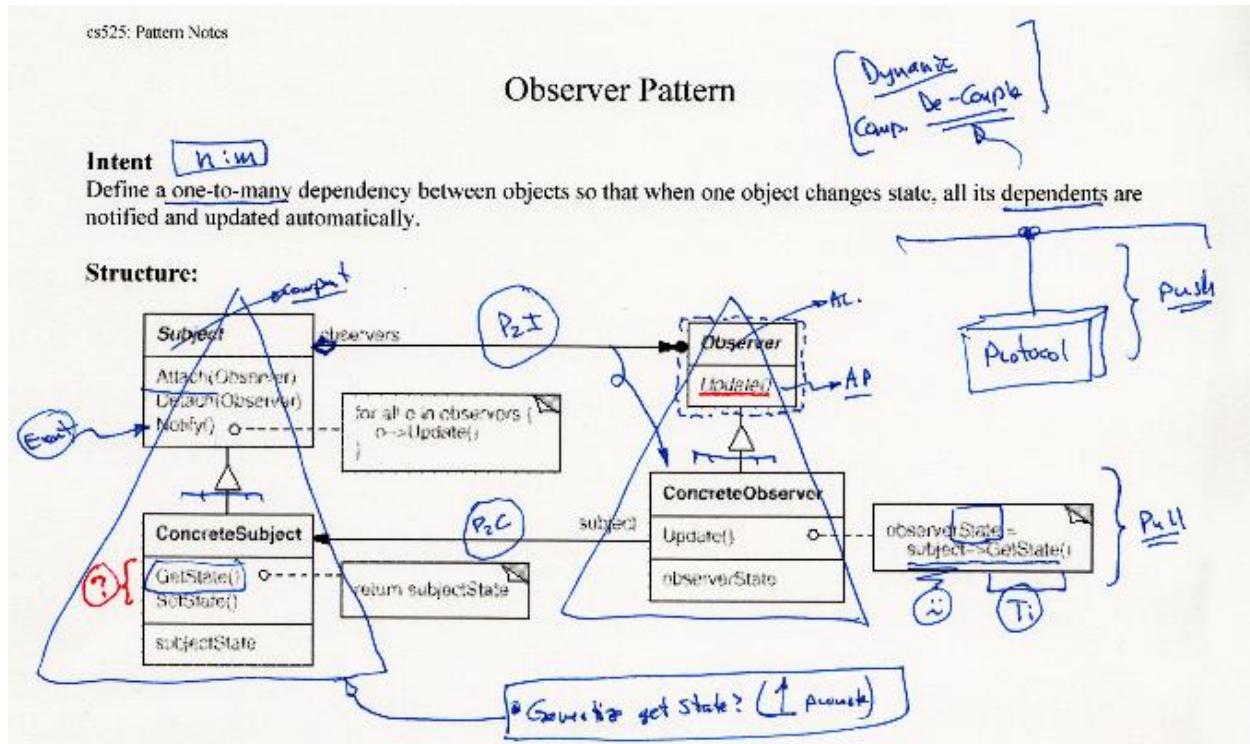
The Java Programming Language
Arnold & Gosling (p.87-88)

- In the Java library, the Observer pattern implementation has some issues
- The Subject is not an interface, so cannot be easily combined with any other classes for inheritance since Java only has Single-inheritance.
- Thus one has to use “Fake MI” using only SI

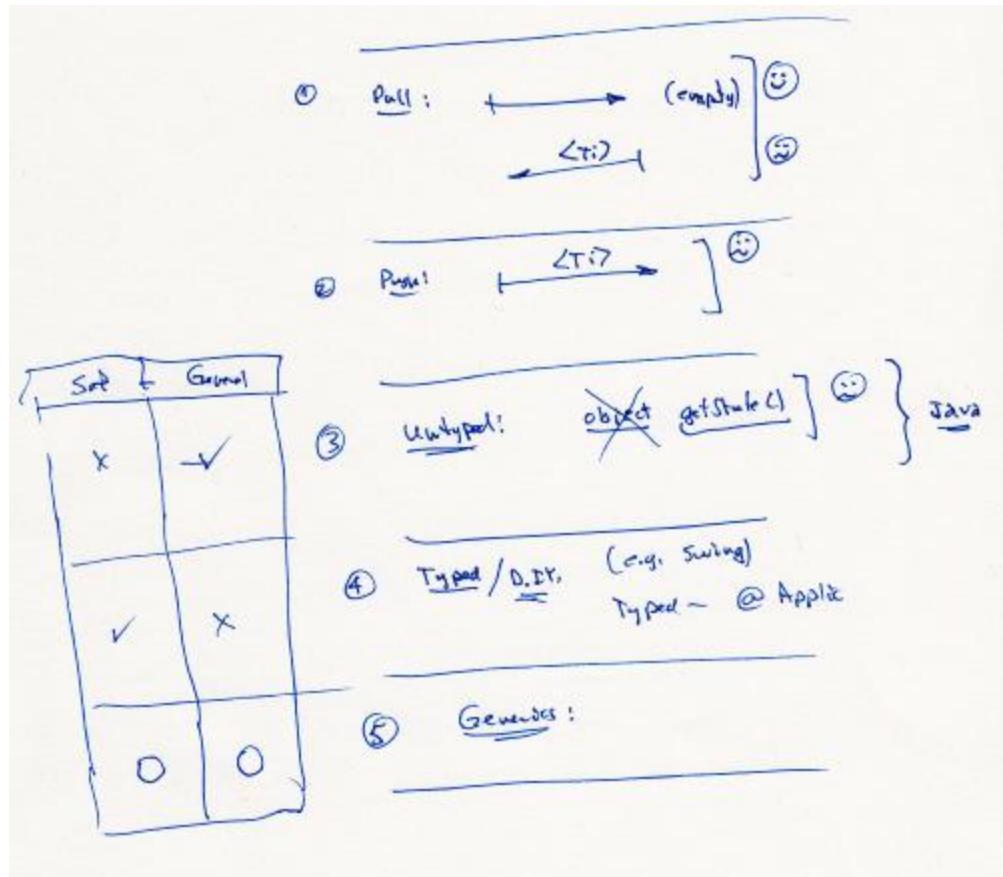
Benefits:

- Decoupling.
- Dynamic Relationship.
- modularity ...

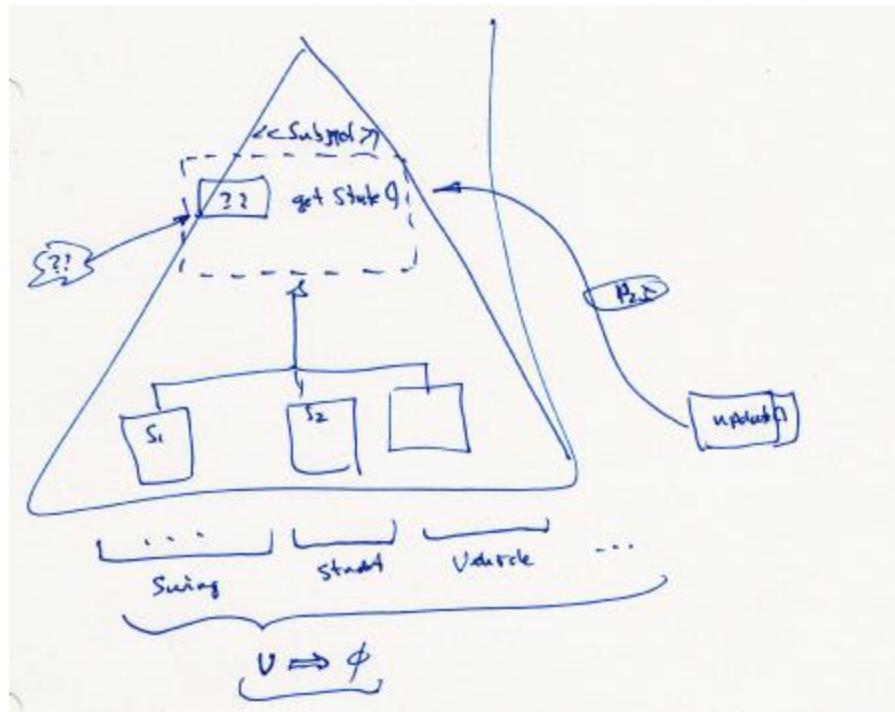
- (n:m) use of the Observer pattern



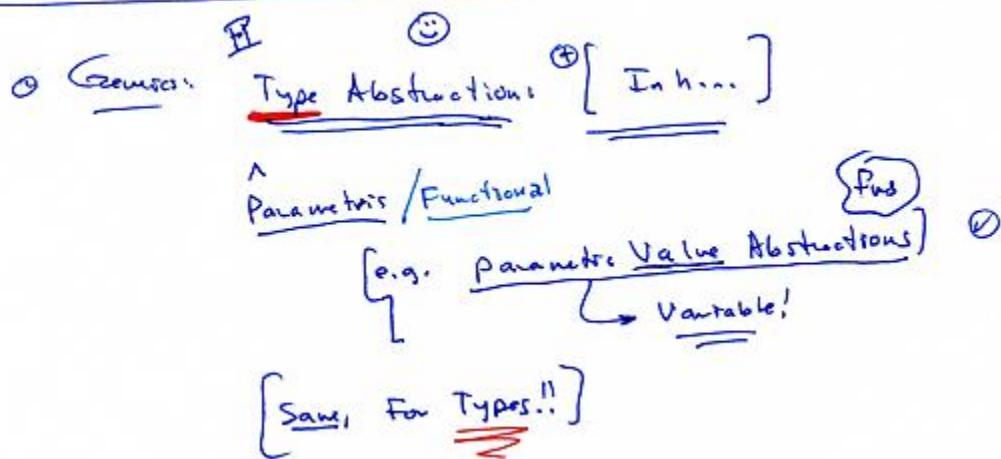
- GOF Definition of Observer Pattern



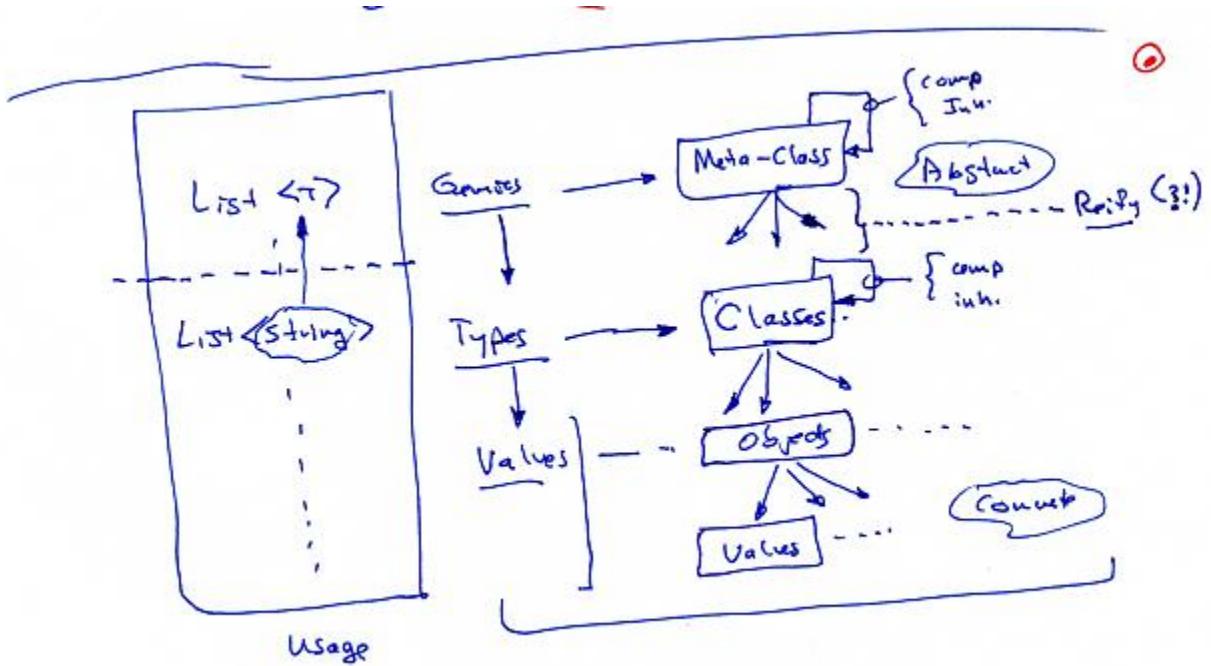
- A main issue with Observer pattern is the bindings – P₂ C relationships. Several solutions are possible.



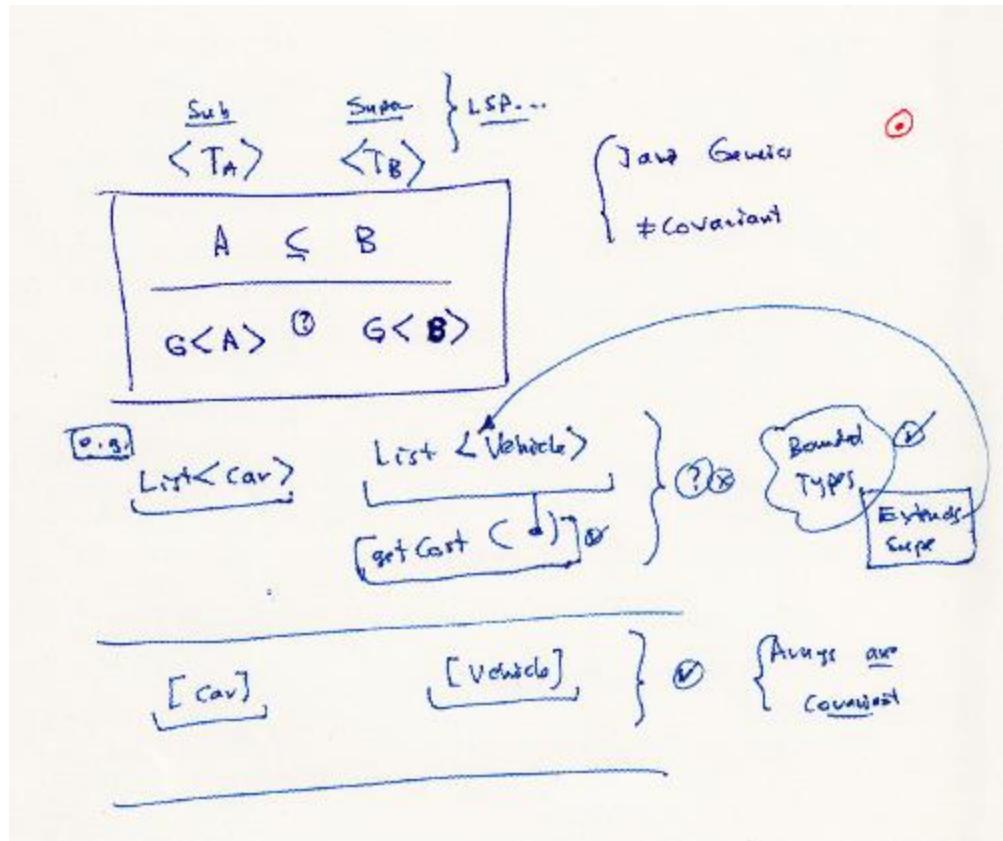
- One idea is to promote (refactor) the `Observable::getState()` method into the superclass of `Observable` – but that won't work for all uses.



- Generics are a solution to the type coupling problems of the `Observer` class



- Generics are Type Abstraction
- One level more of abstraction in the design hierarchy



- Java Generics are not *covariant*

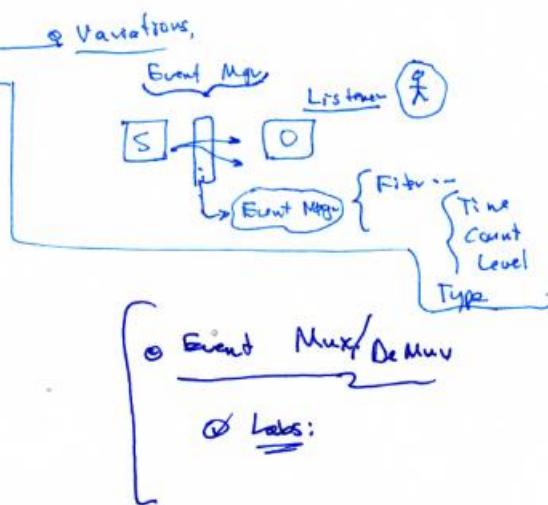
Day 4: Façade, Strategy, and Template Patterns.

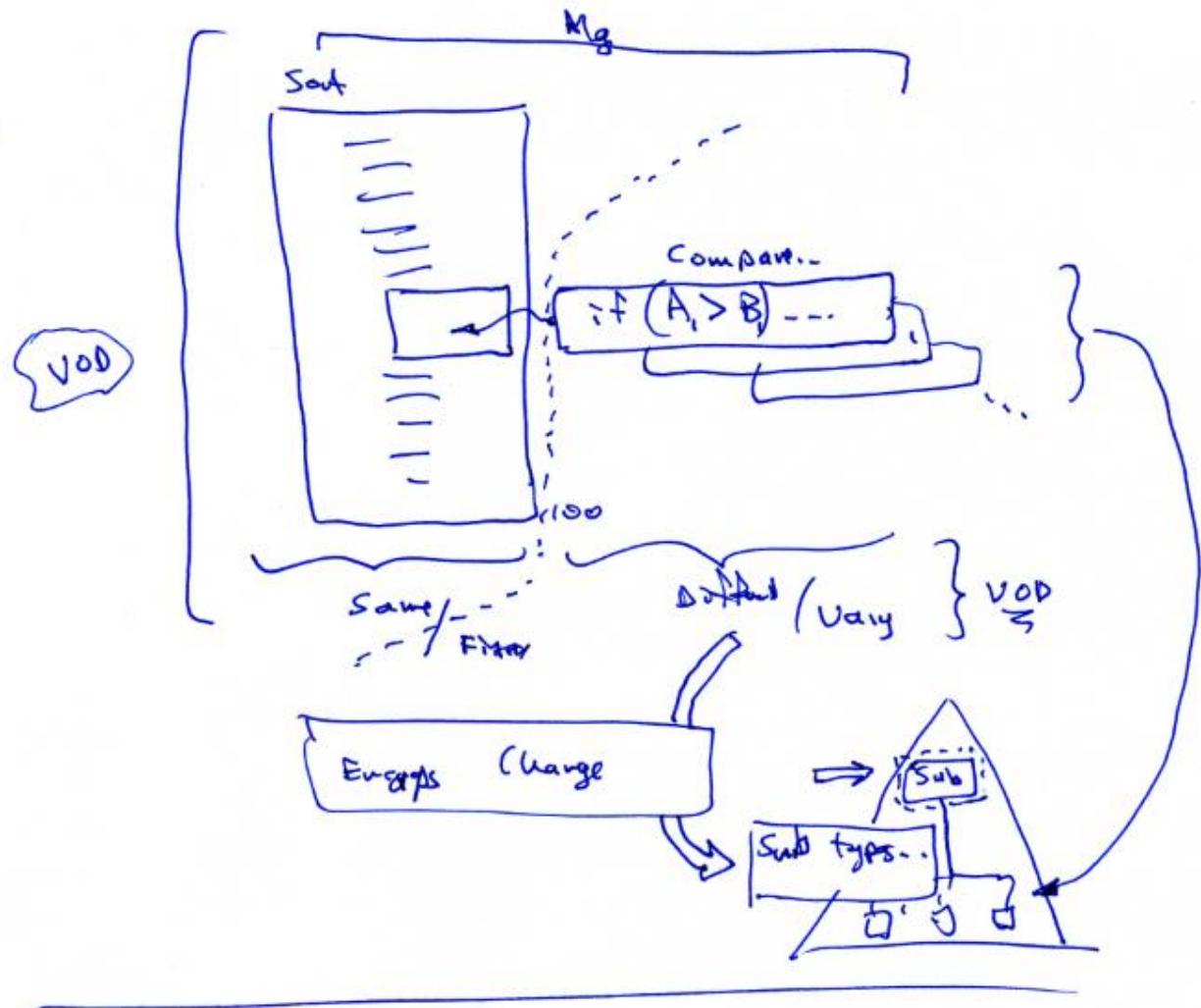
Advanced Software Development

Daily Topics

Day 4:

- Quiz & Review
- Observer
 - examples & issues
 - Java Implementation \in SI \Rightarrow MI
 - Types, coupling (push, pull)...
 - generics
- Façade pattern
- Strategy pattern
 - uses *functor*
- Template pattern
 - \approx refactored Strategy
- Lab:
 - Strategy Lab

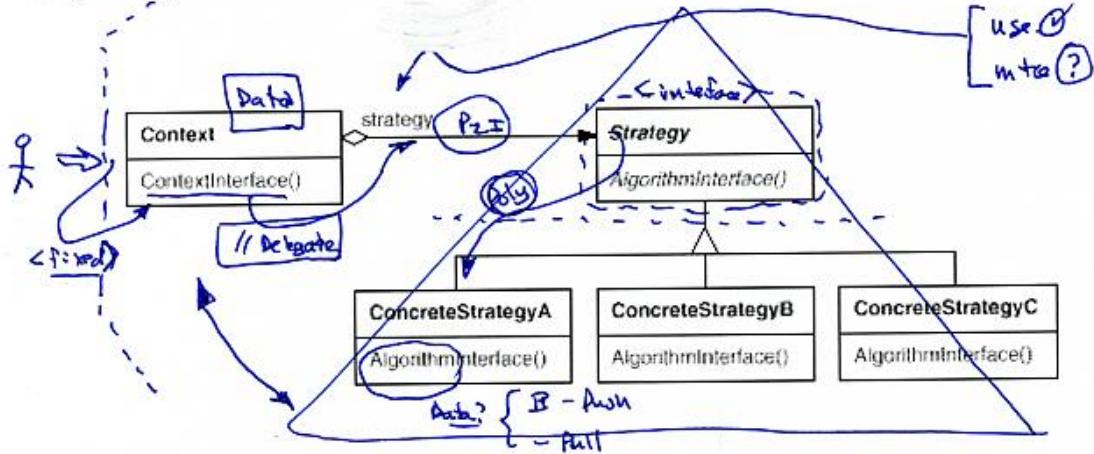




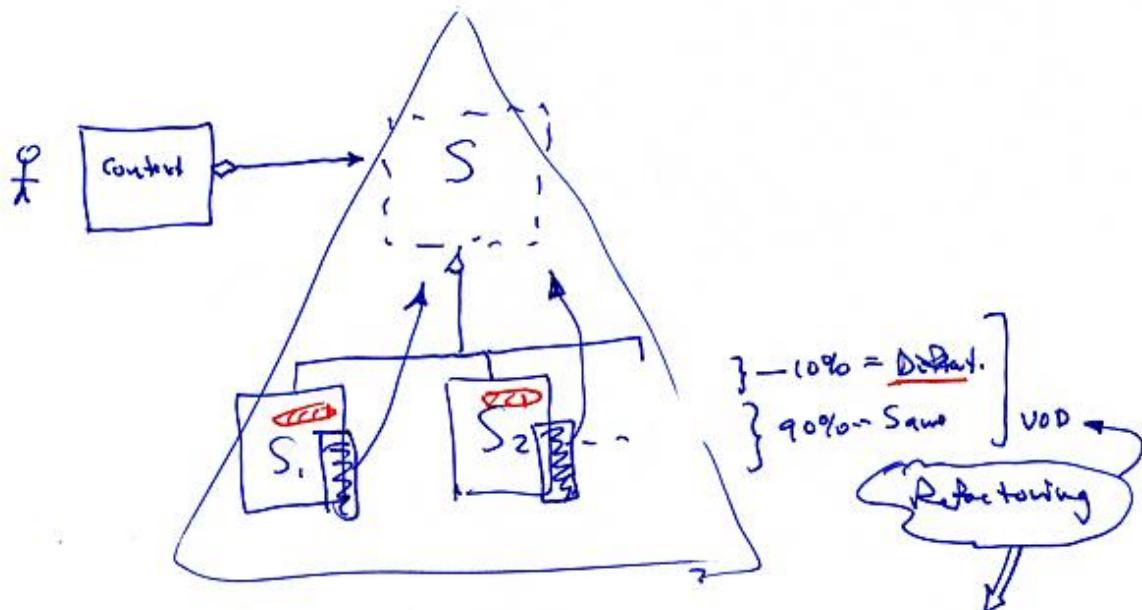
- The strategy pattern uses VOD to isolate and encapsulate changing parts of an algorithm.

Strategy Pattern

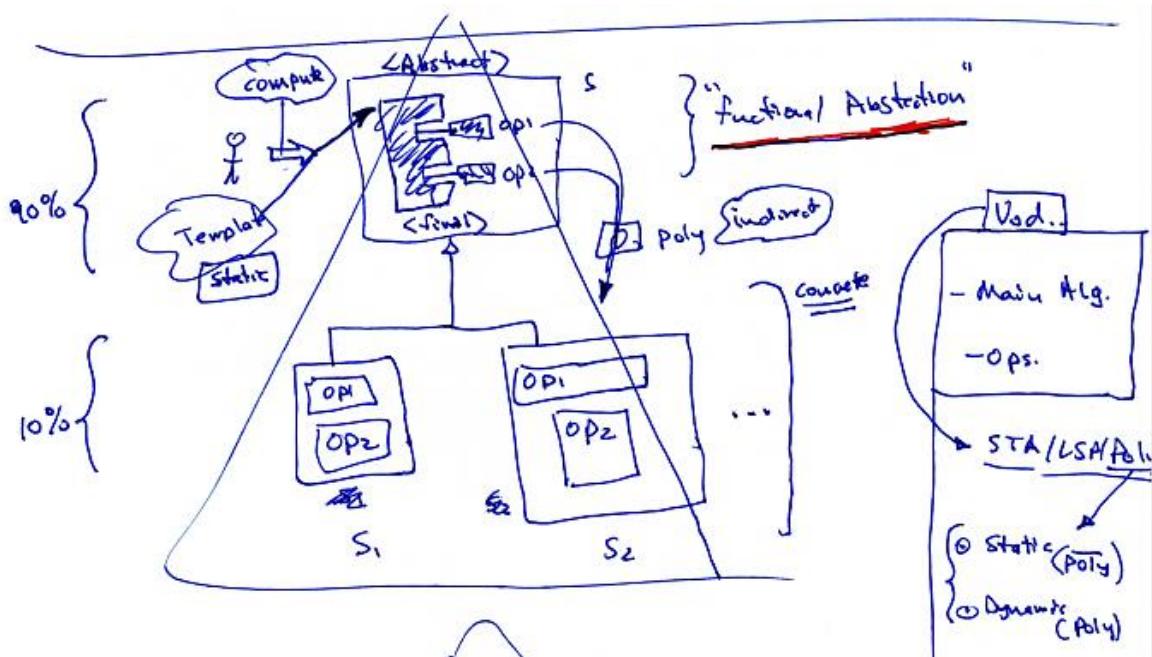
Intent: Inh.
 Define a family of algorithms, encapsulate each one, and make them interchangeable. Useful when you want to do the same thing, but perhaps in several different ways. Strategy lets the algorithm vary independently from clients that use it.



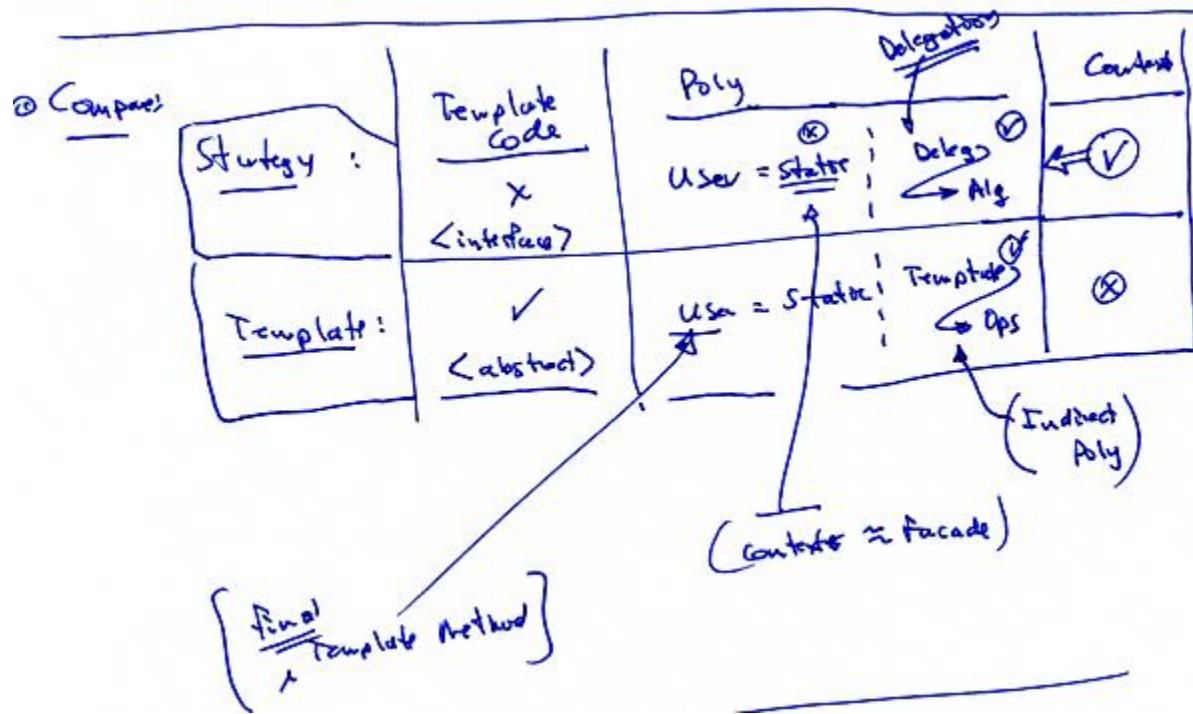
- Strategy pattern from GOF



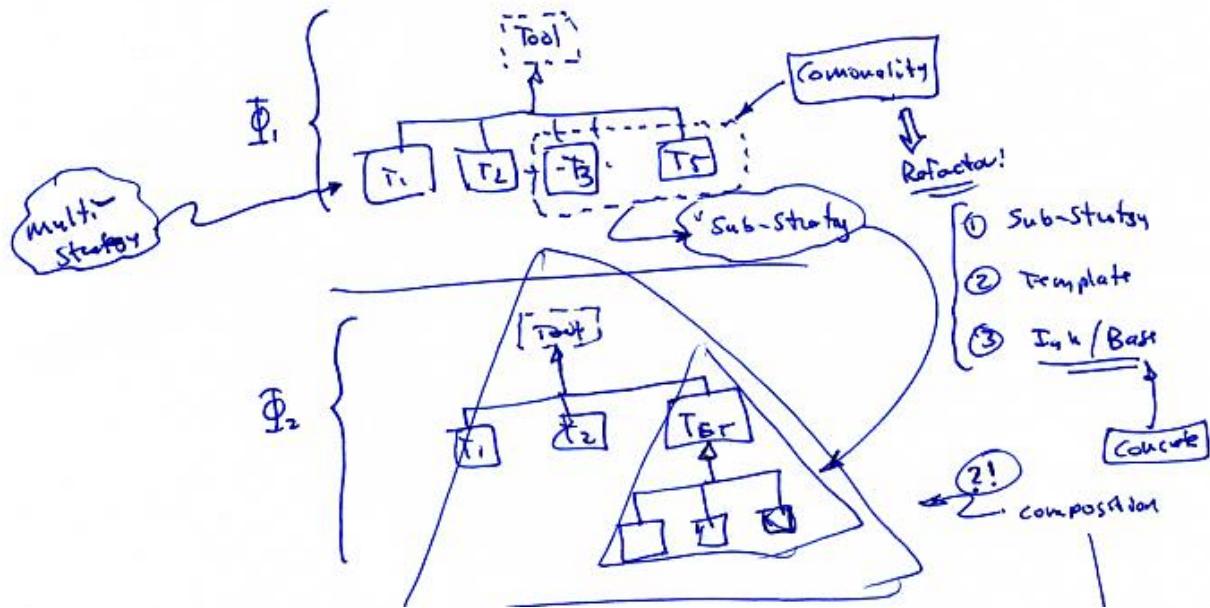
- In some cases a Strategy pattern can use refactoring, to eliminate common code in the concrete strategies



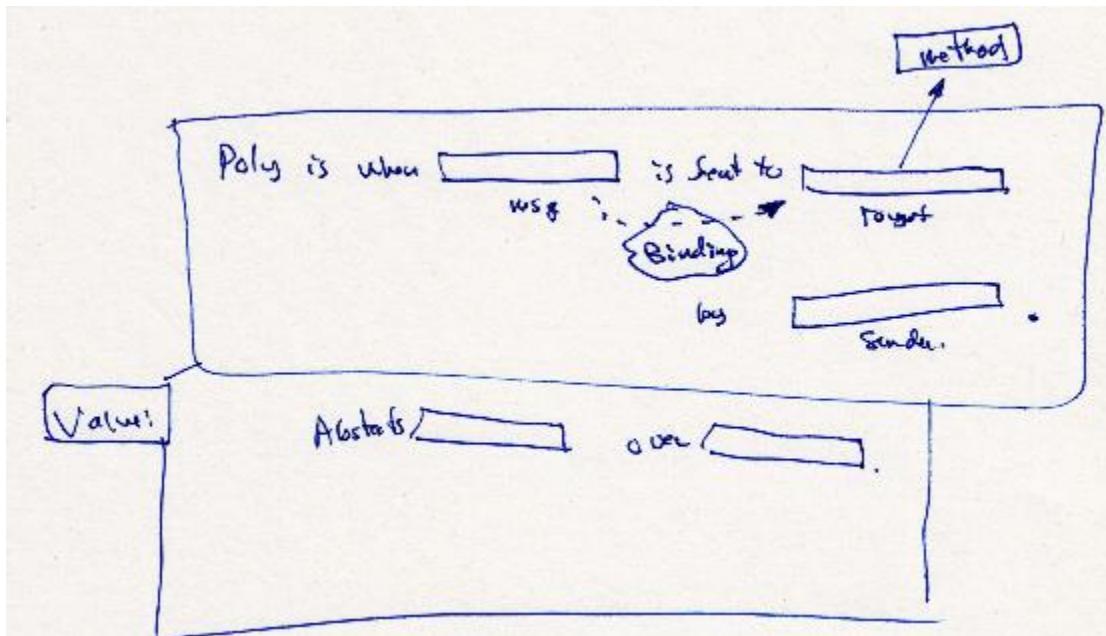
- This leads to the Template method pattern



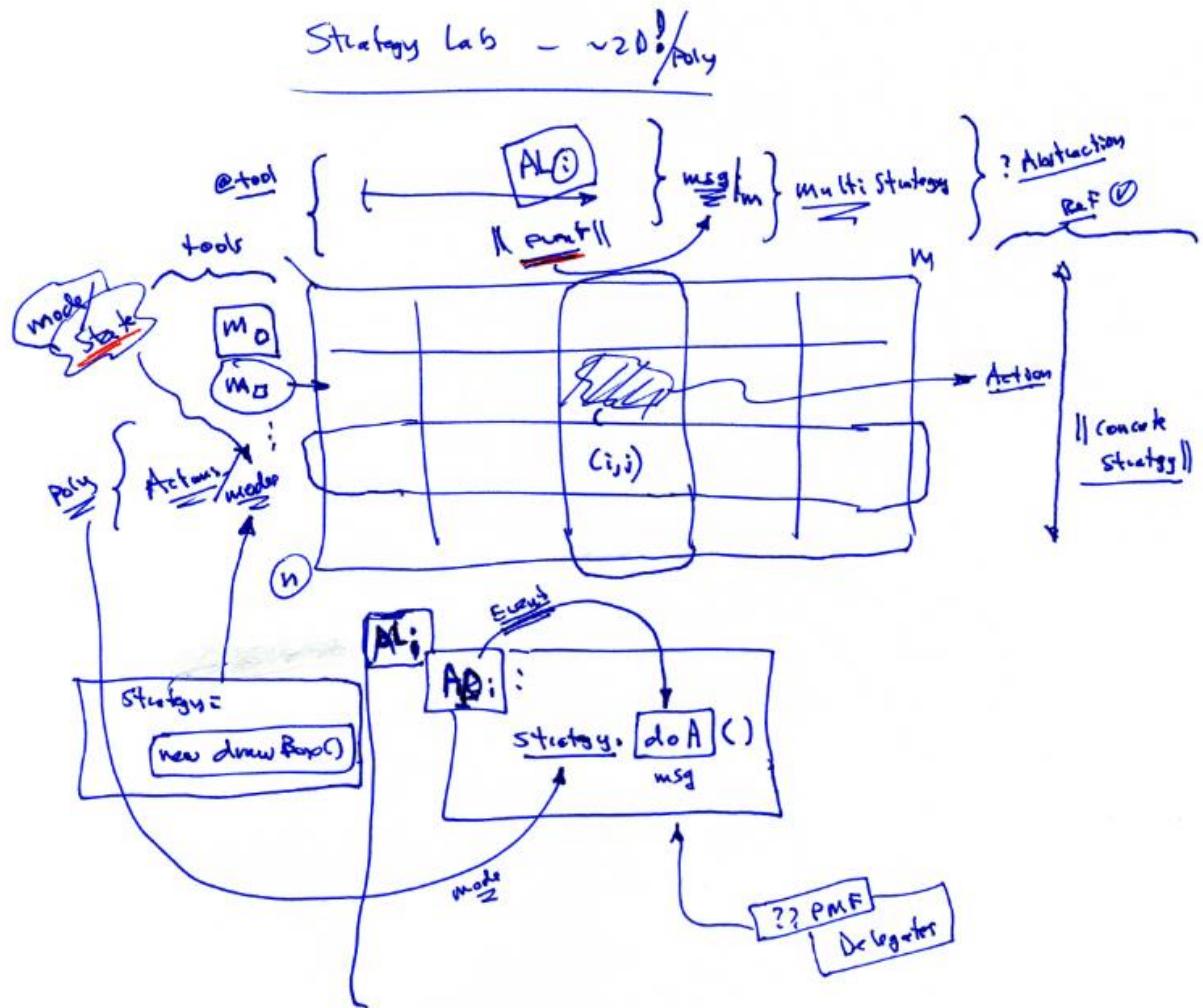
- Comparing the Strategy & Template patterns



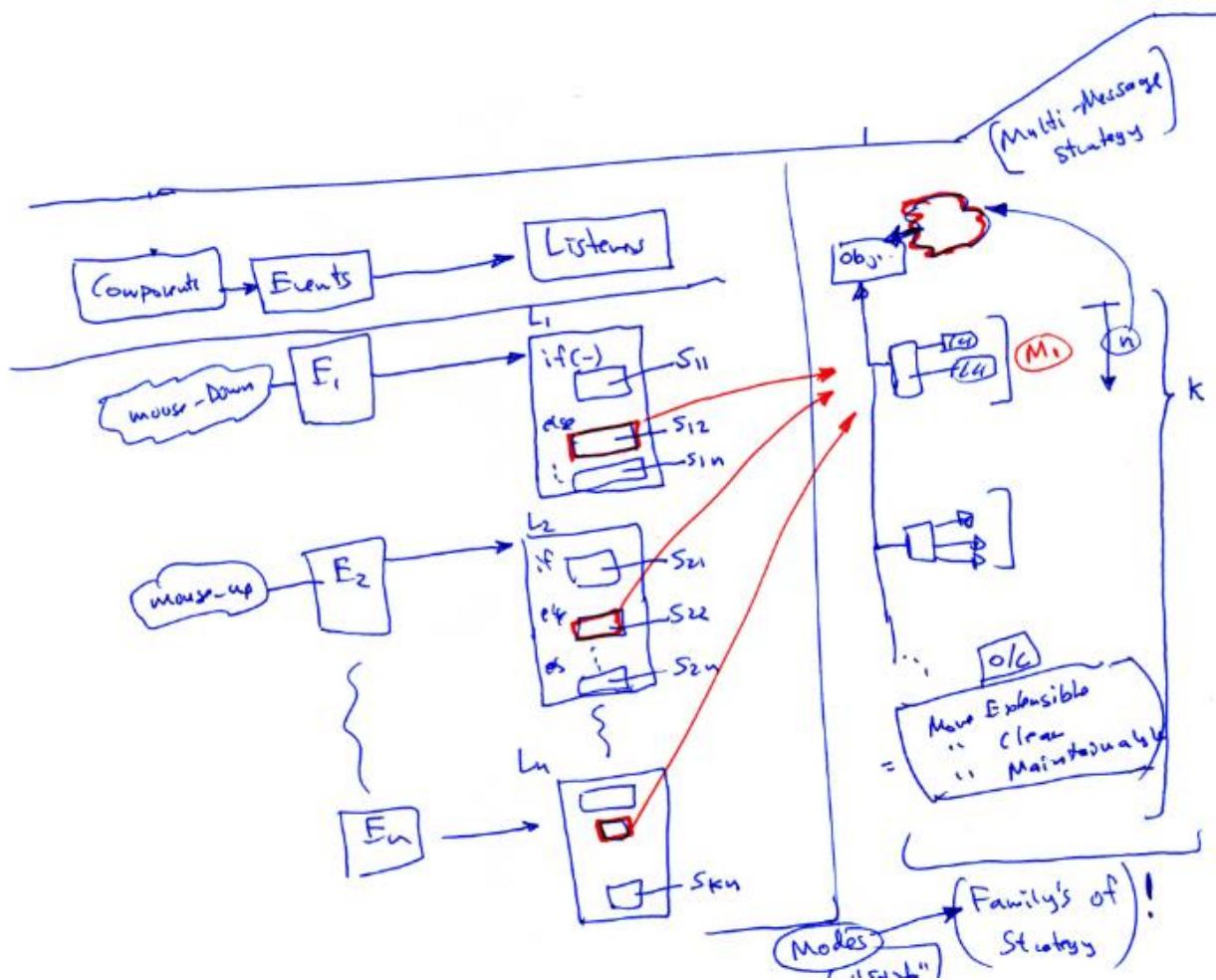
- Strategy Lab



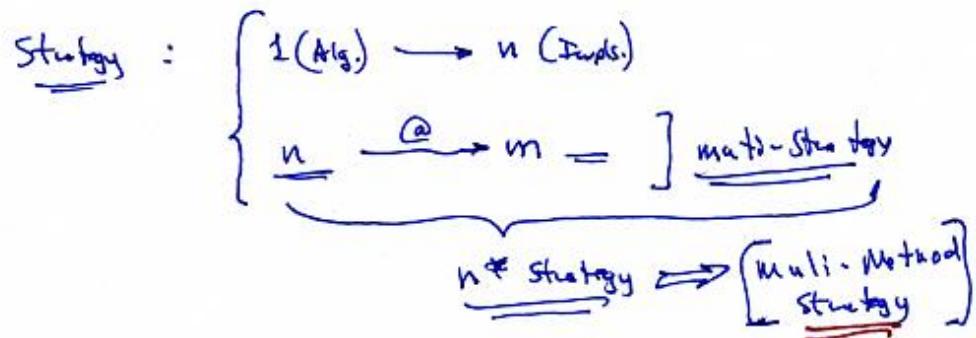
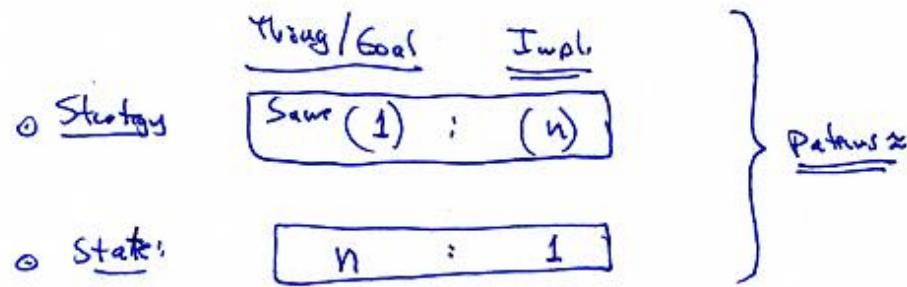
- We need to be precise when describing how/where/when/why polymorphism is used.



- Strategy Lab \approx 2D Poly!

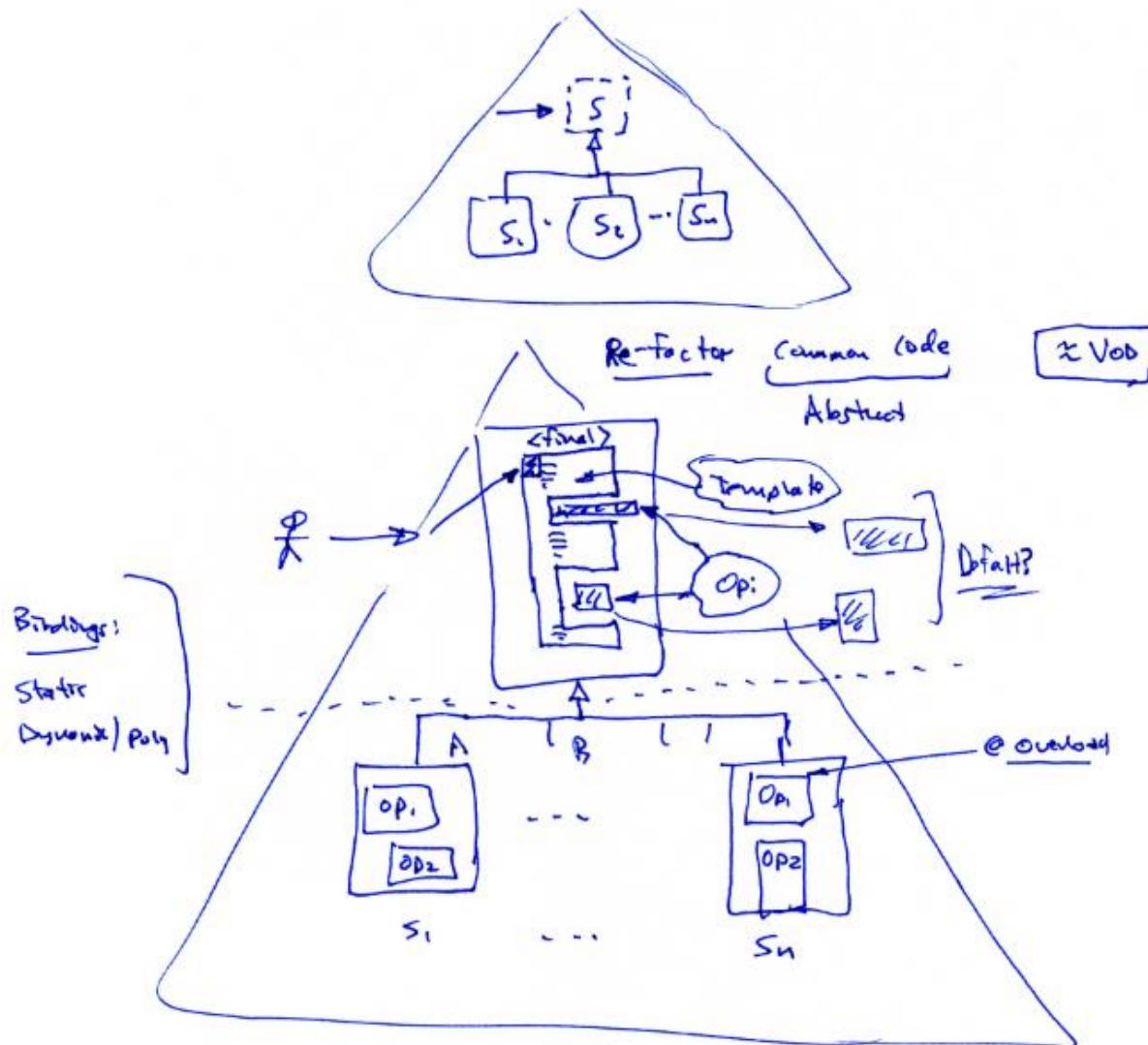


- Strategy lab is mostly just refactoring, moving code around



- Comparing state & strategy patterns

① Template = Refactored Strategy



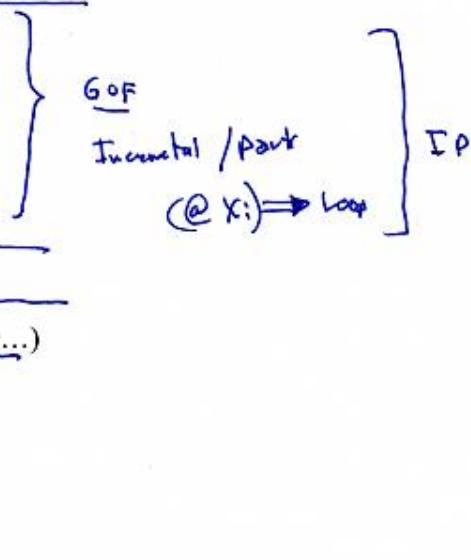
- Template ≈ refactored Strategy
- However it does not (typically) include a context – it has a different focus of improvement (the refactoring)

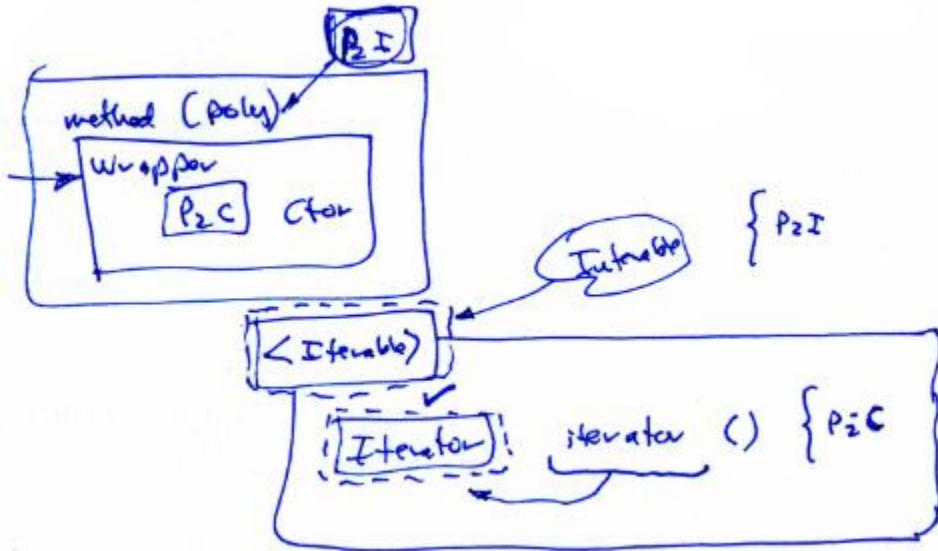
Day 5: Iterator Patterns.

Daily Topics

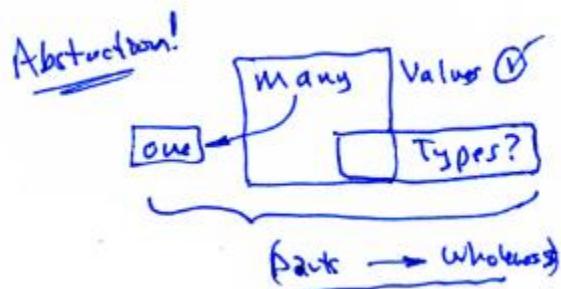
Day 5: Collections & Iteration

- Quiz & Review
- Collections
 - Java collections \Rightarrow generics
- Iterators
 - issues: Robust, modular, stateful, ...
 - polymorphic iterators
- Iterator pattern: GOF
 - general design \Rightarrow OO
- Iterator factory (pattern)
- Variations: (from @element \Rightarrow wholeness...)
 - uses *functor* pattern
 - (Iterator + functor) \Rightarrow ...
 - internal iterator
 - selective iterator
 - \approx Functional Programming
 - higher level of abstraction
 - Declarative, less over constraint (sequential)
 - Other fanciness...
- Soon, Java *closures* will make iterators more powerful, and popular!

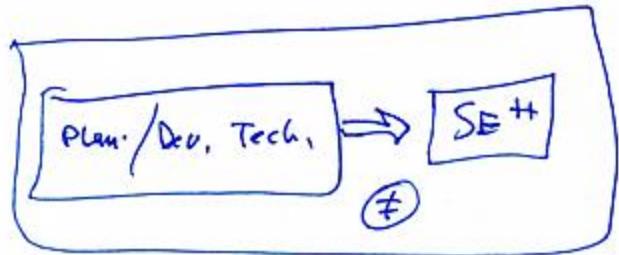




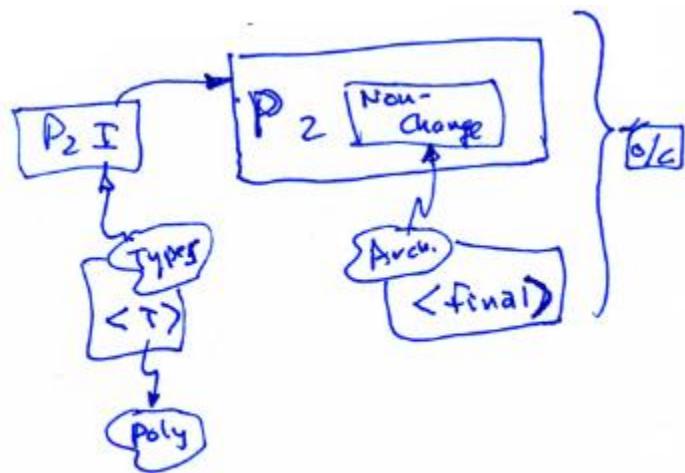
- Quiz & Review:
 - Factory methods encapsulate a P\2C constructor,
 - In a P2I method



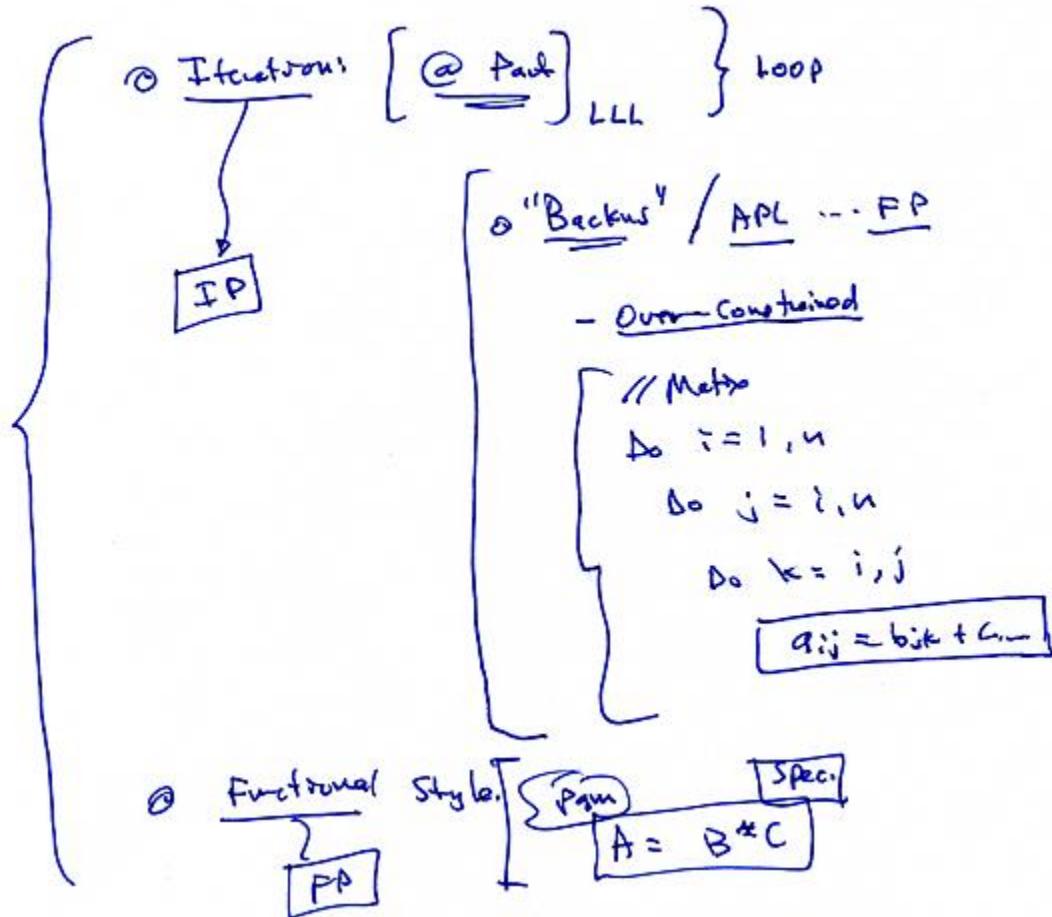
- Collections are an important abstraction;
 - Wholeness of parts;
 - Parts by values (elements)
 - And by types of elements (STA)



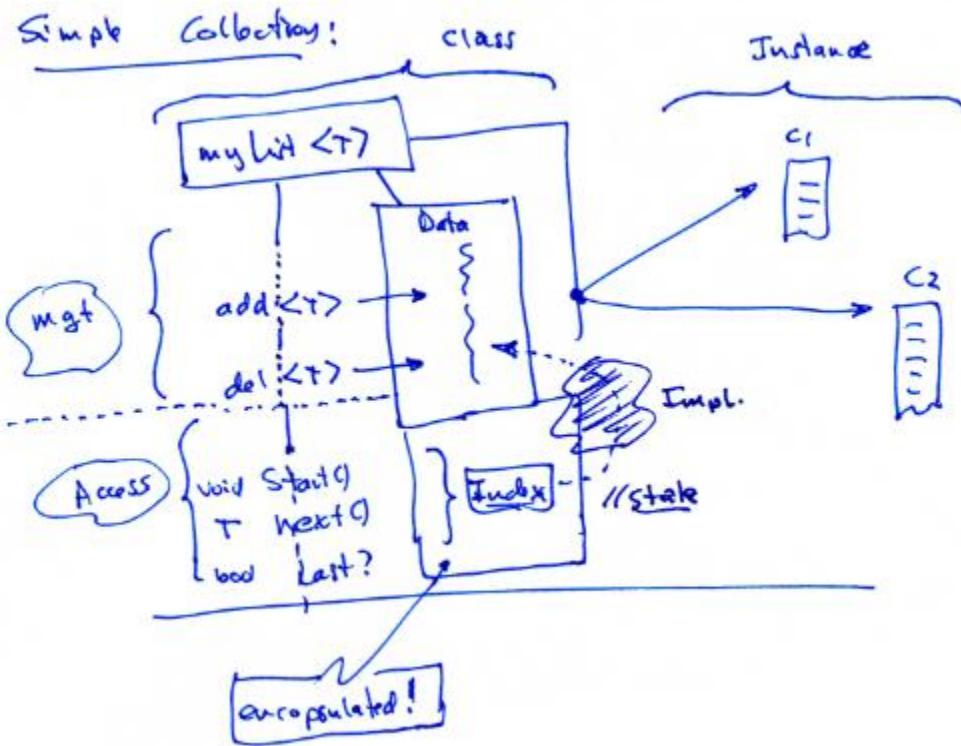
- Quiz: Reminder, Development techniques (Dynamic binding, modularity, Poly) are not the same as SE benefits (extensibility, open/closed, ...), which should each \Rightarrow \$\$\$ business benefits.



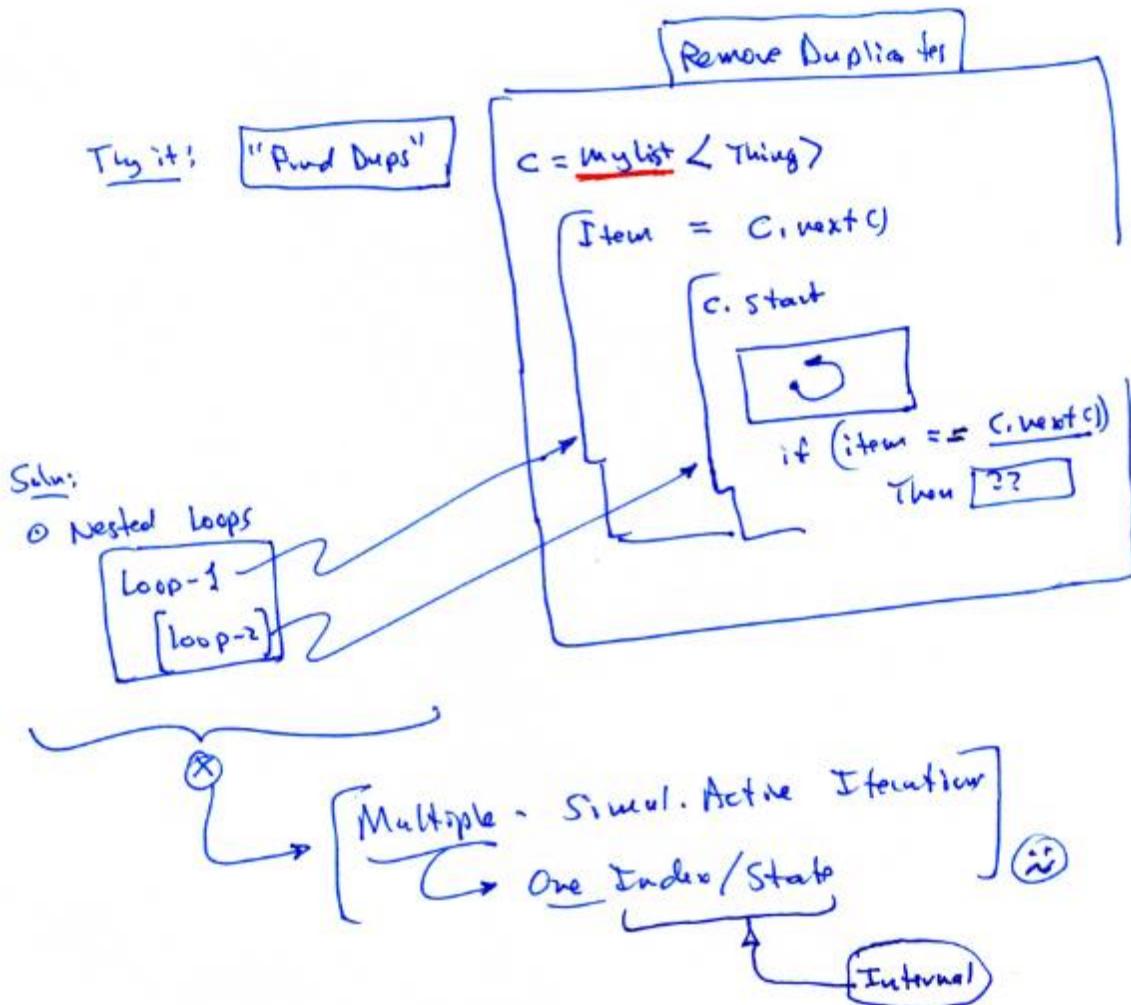
- There is an additional application of Open/Closed principle;
- Poly is the main one we have studied; *fixed interface*, varying poly concrete sub-types
- Another would be architectural – making a method *<final>* (*static binding*), so that it cannot be changed, but it can delegate to a ($P_2\text{ I}$) reference (e.g. Strategy pattern).



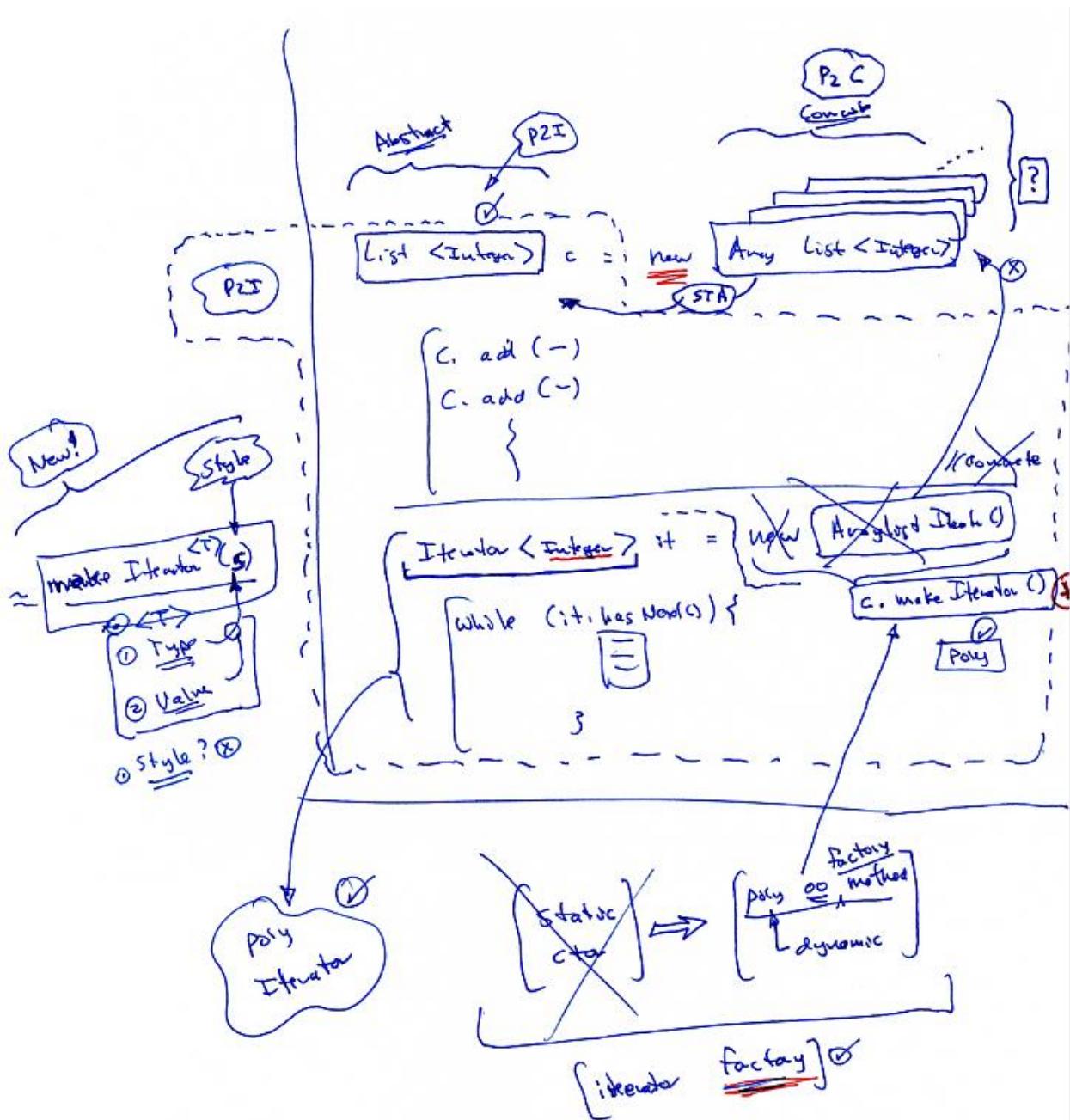
- Iteration in IP languages is too *low-level* (semantics)
- \Rightarrow *over constrained*



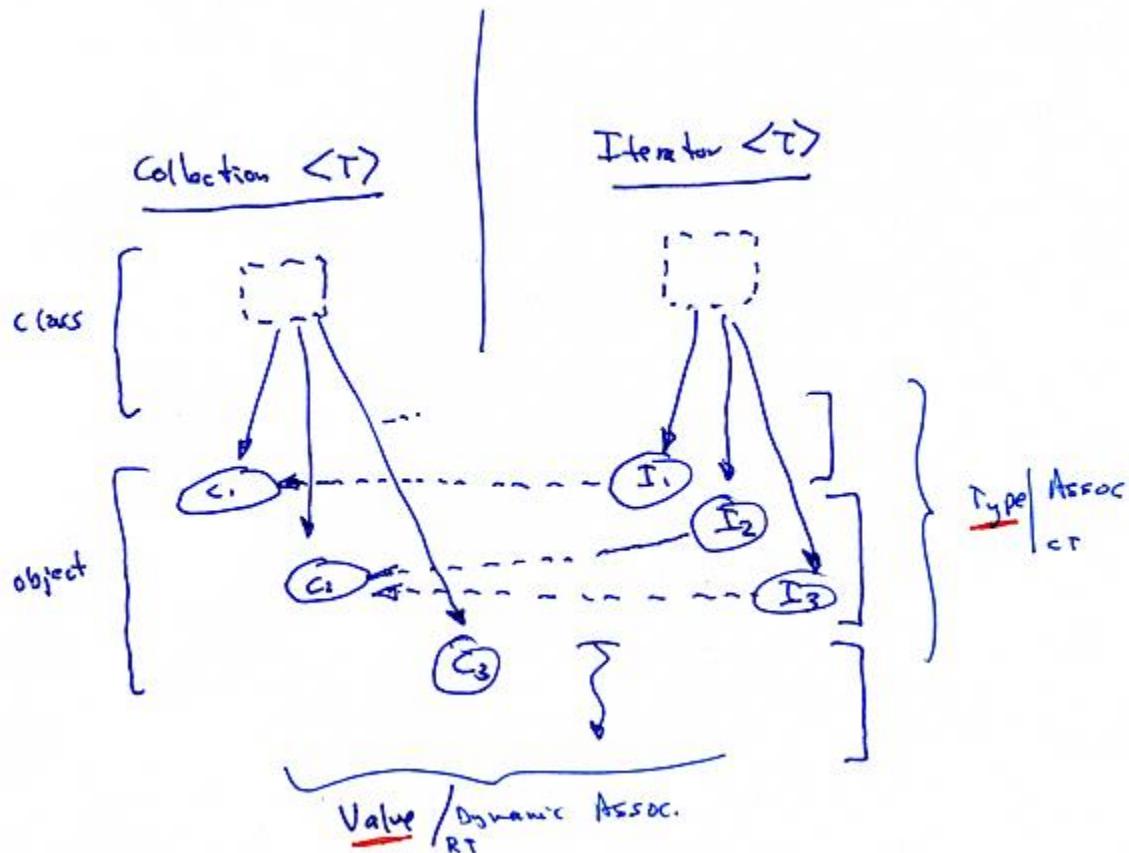
- Example of a simple collection implementation
- With some access (iteration) added



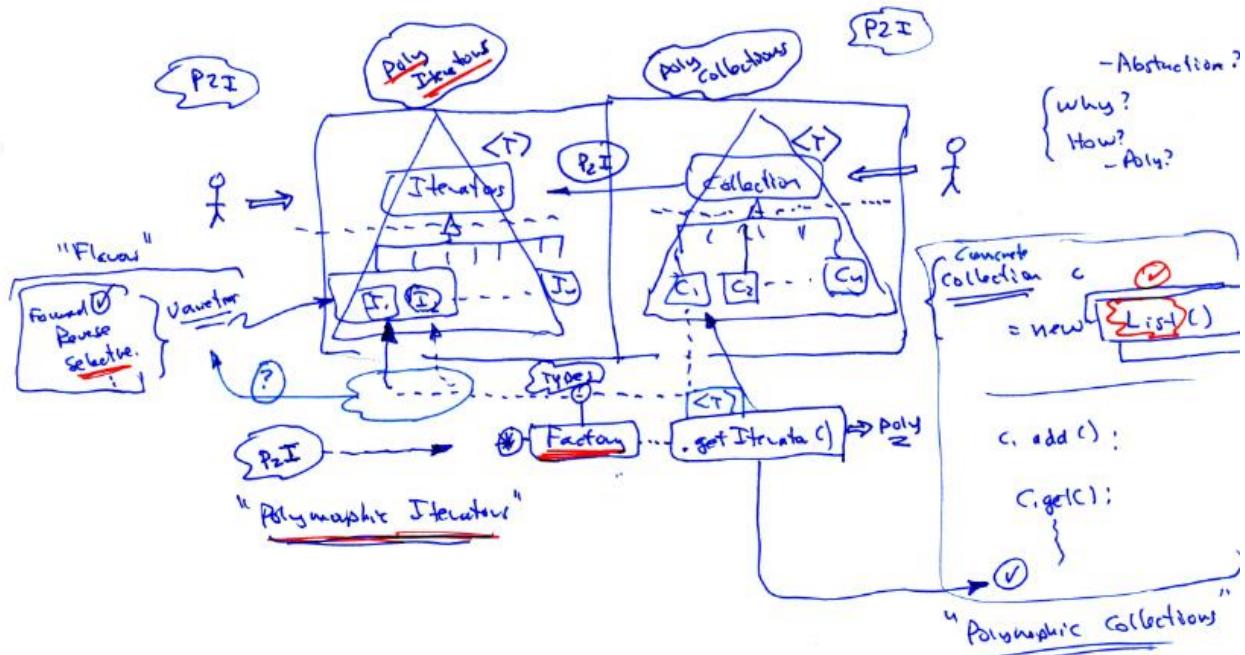
- Testing it – fails!



- Using P2I for iteration



- Multiple associations between iterators & collection – both static (type, CT) and value (dynamic, RT)



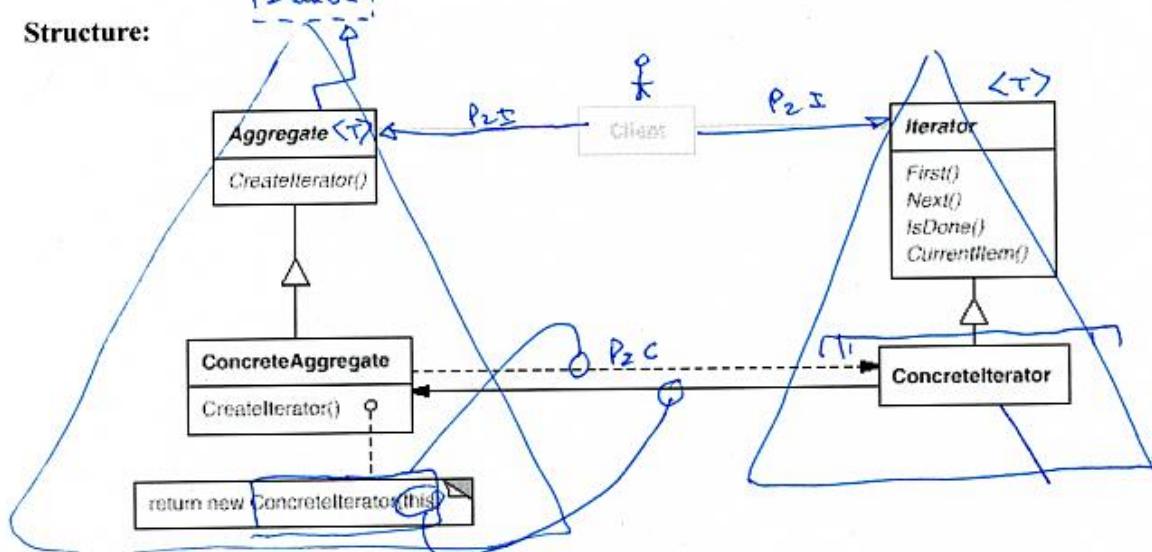
- Dual hierarchies of iterators and collections
- Related by type, and value

Iterator Pattern

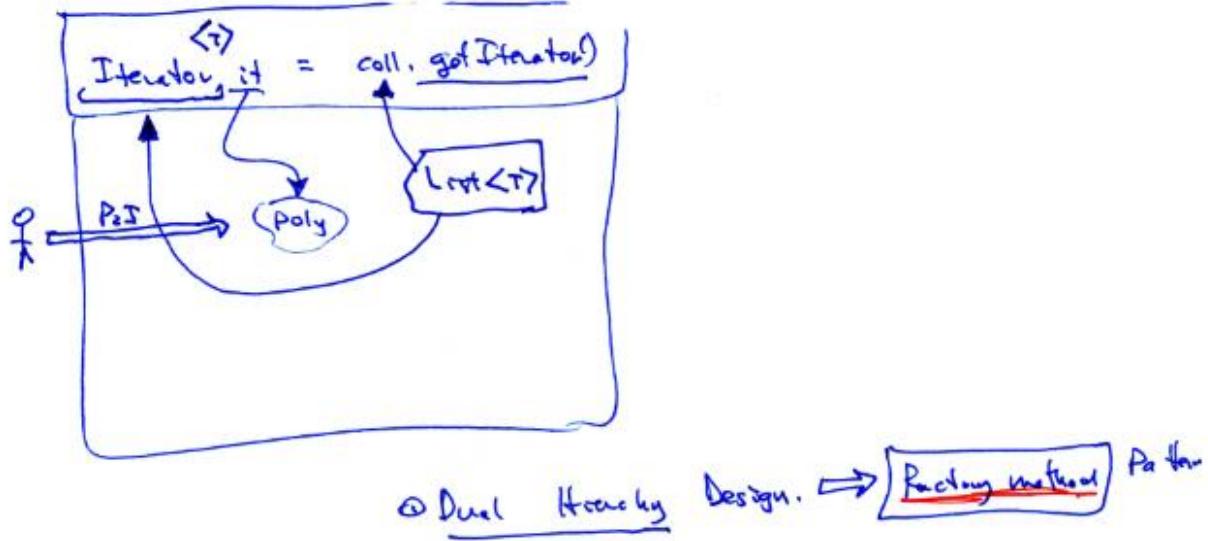
Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

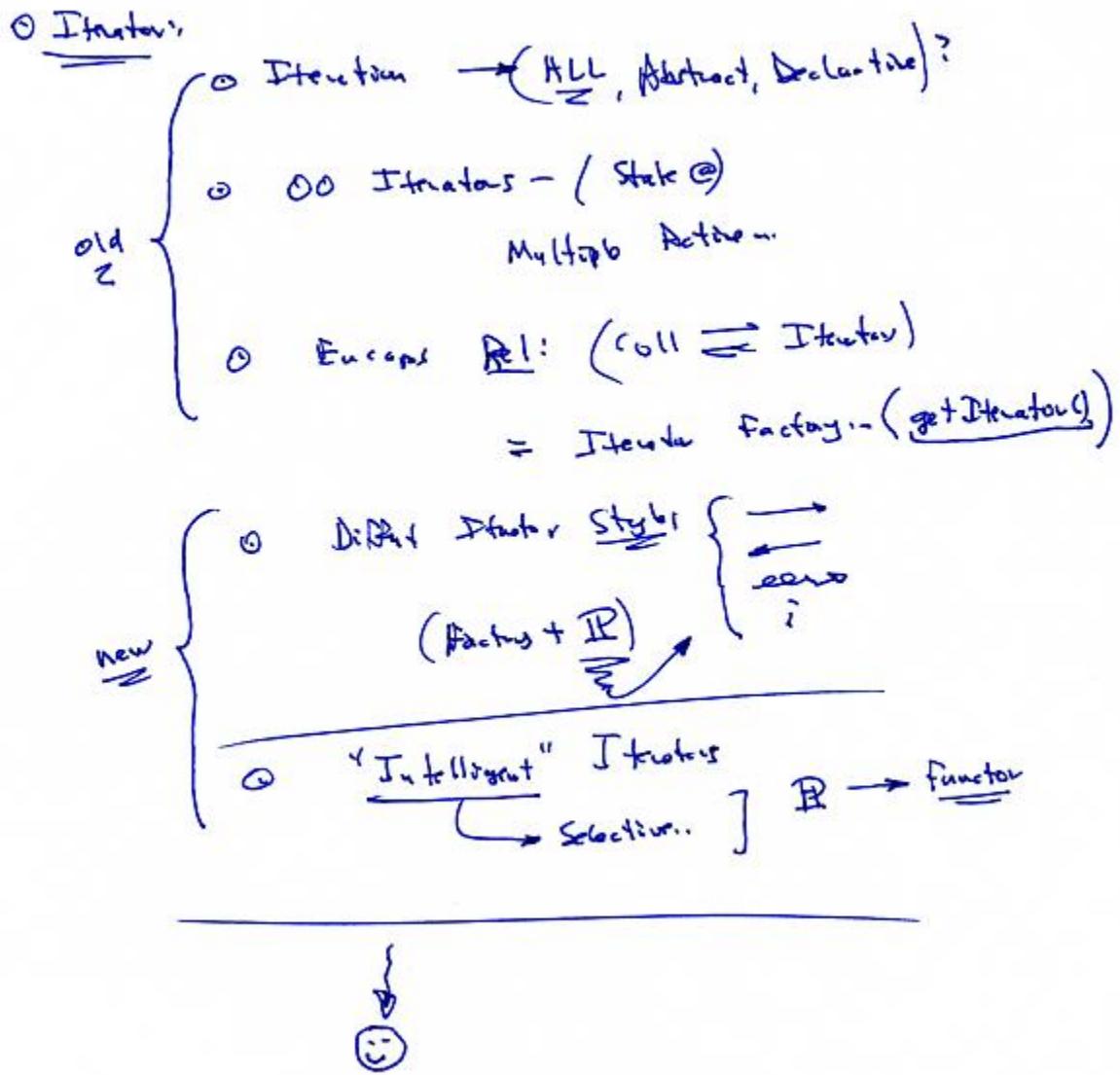
Structure:



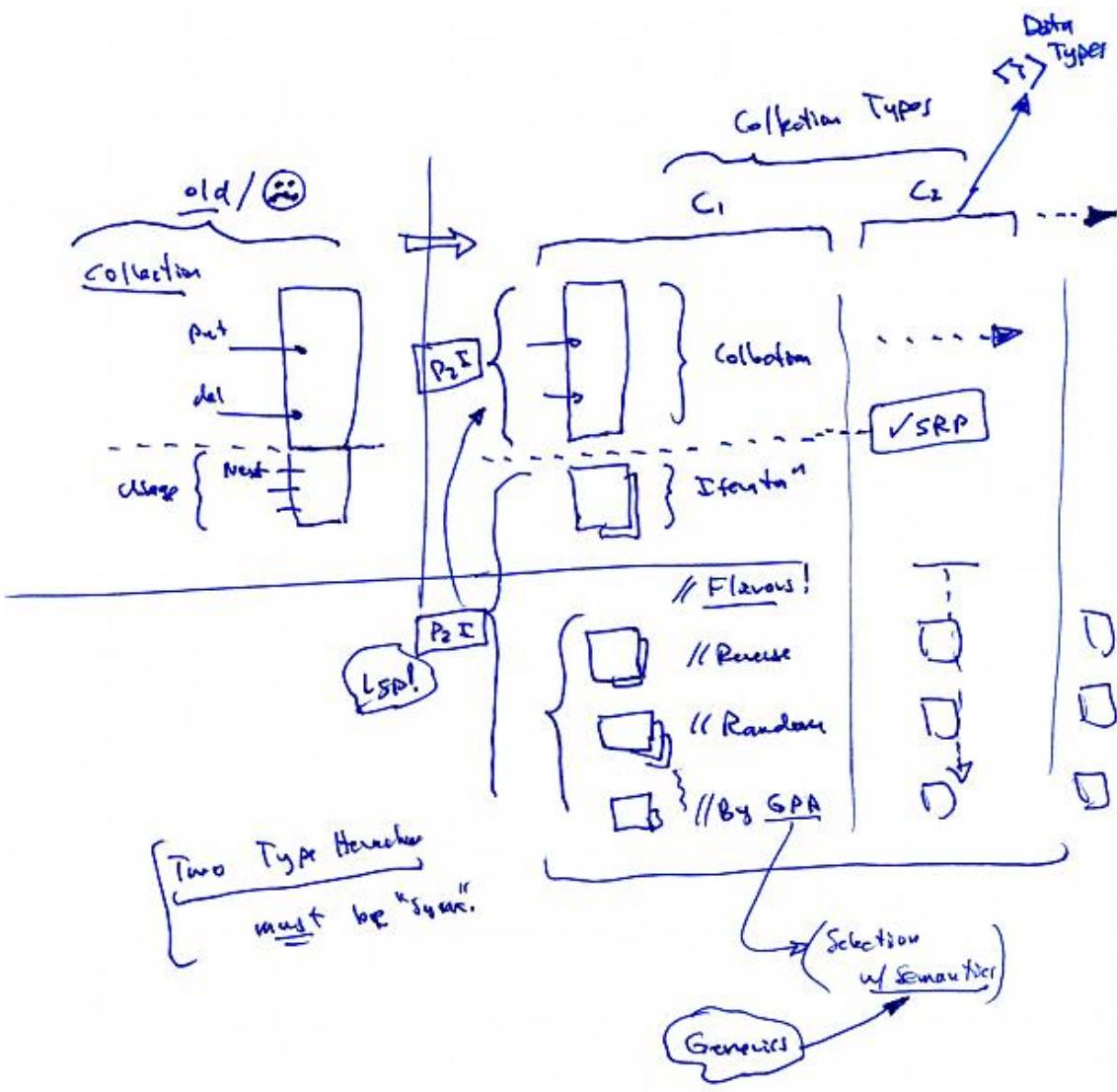
- GOF Description



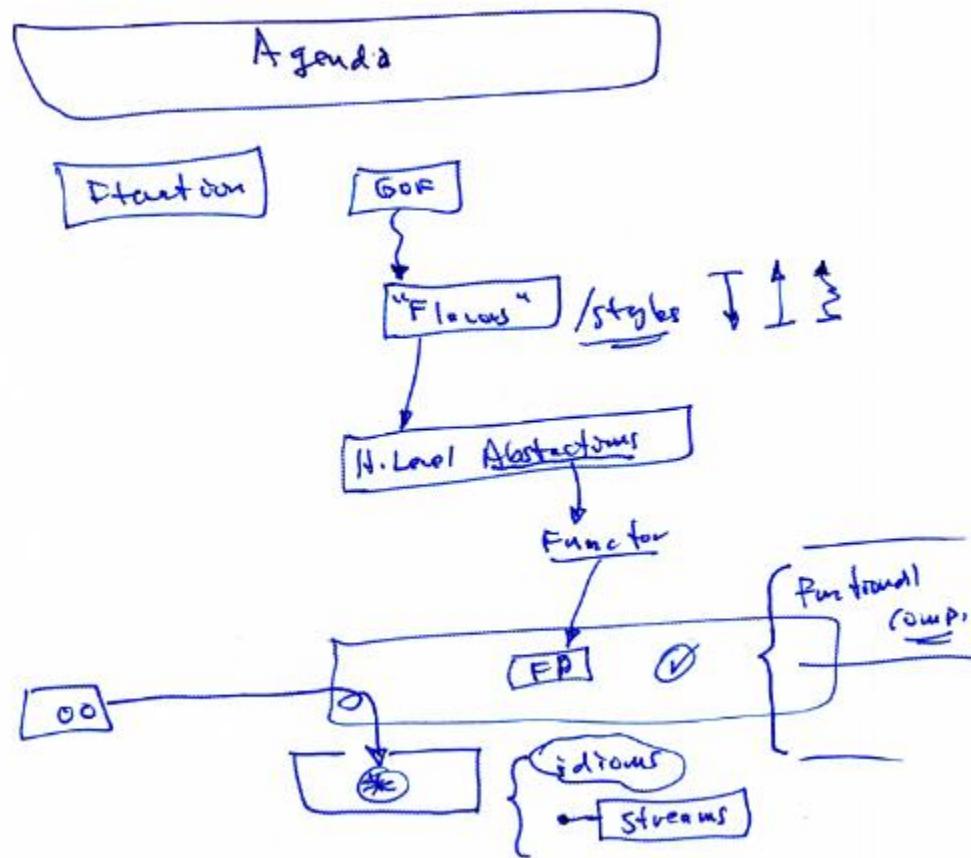
- Factory method for “`getIterator()`” instead of static constructor – concrete type



- Now we add fancy (smart) iterators, and other forms of iteration



- Varieties of iterators



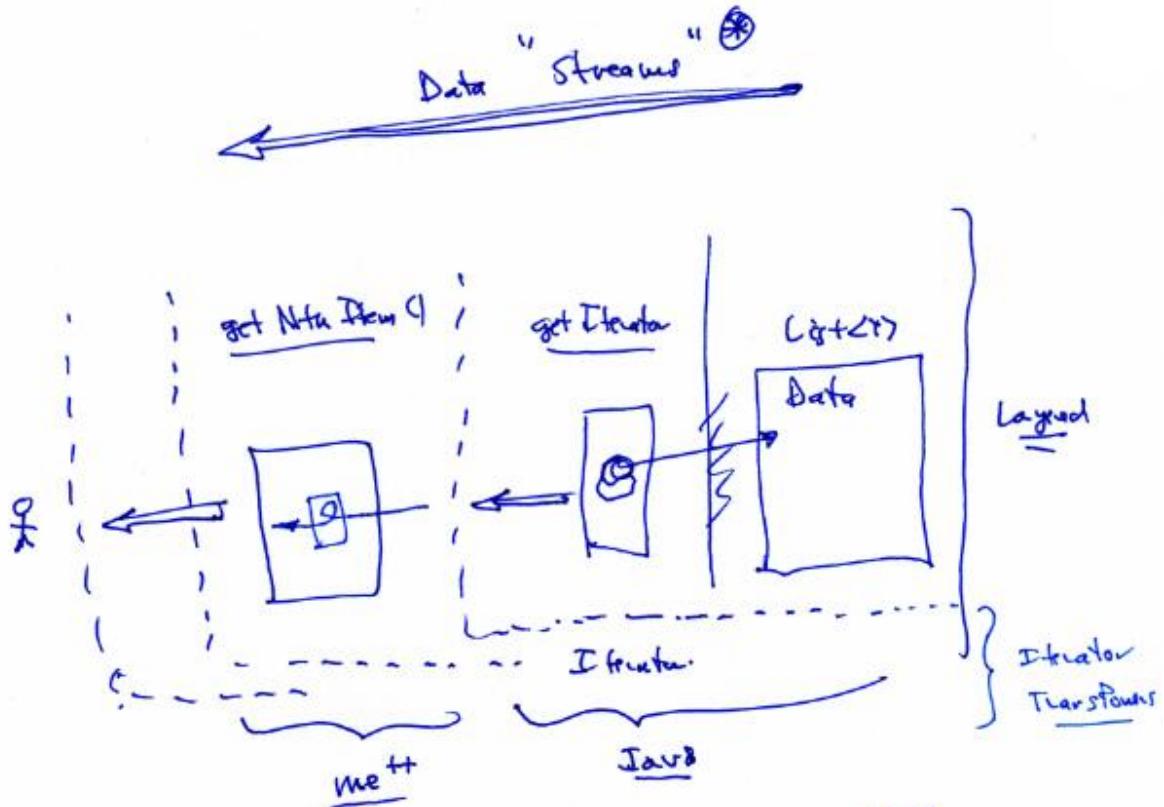
- We will extend GOF iterators with higher level (functional) idioms

cs525: Pattern Notes

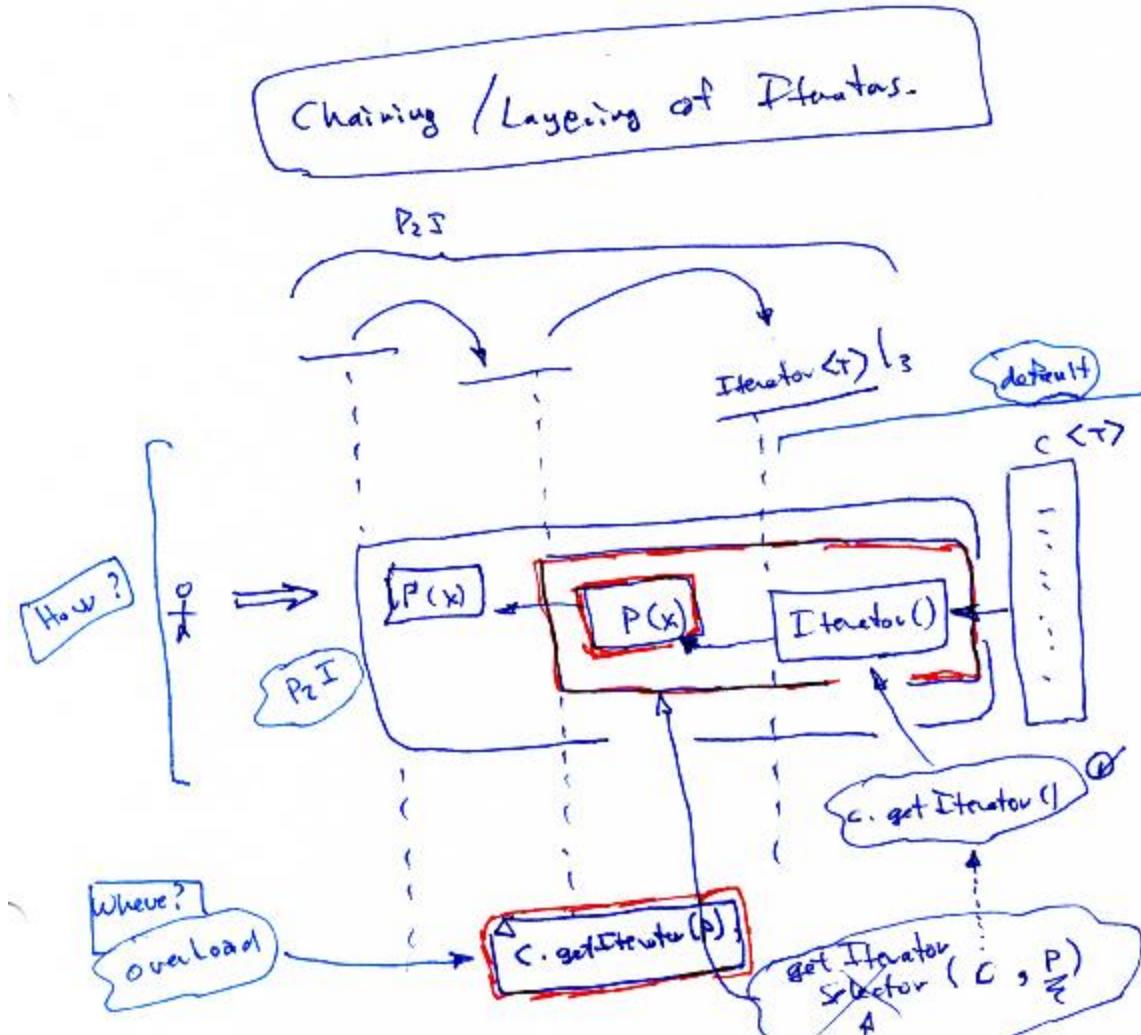
Iterator Styles:

		Iterable	
		Complete Traversal	Selective Traversal
External	$\text{Iterator } it = c.\underline{\text{iterator}}();$ $\text{while } (it.hasNext())$ $\quad \text{ProcFun}(it.next());$ Or, $\quad \text{foreach (T e : c) { ... }}$	$\text{Iterator } it = c.\underline{\text{iterator}}(\text{Pred } p);$ $\quad [\text{Then use normally! } ??]$	P
Internal	$c.\text{doAll}(\text{ProcFun})$		$c.\text{doAll}(\text{Predicate}, \text{ProcFun})$

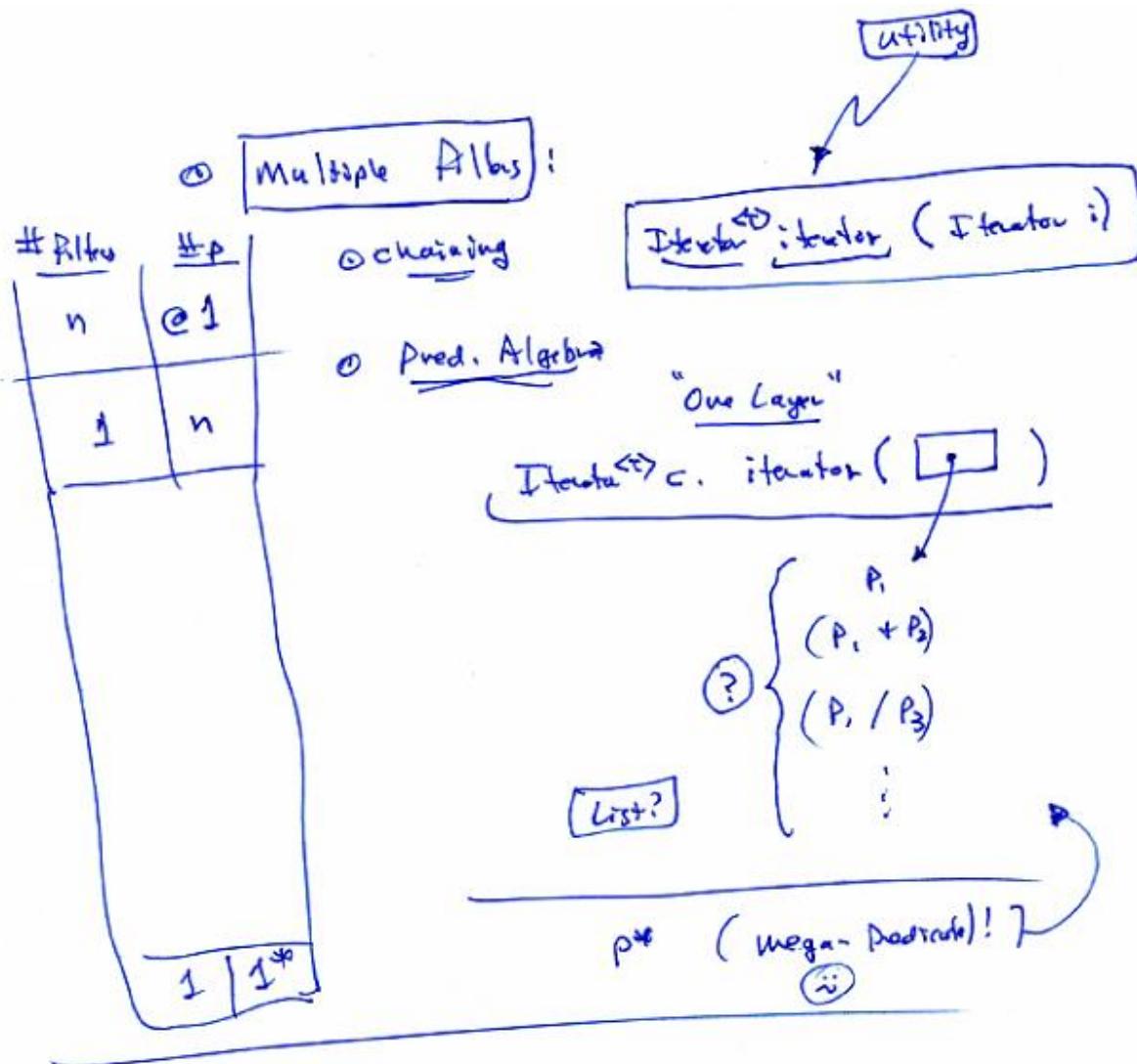
- New iterator types



- Chaining of iterators by levels of P2I gives a new design approach
- *Data streams* (soon in Java-8!)

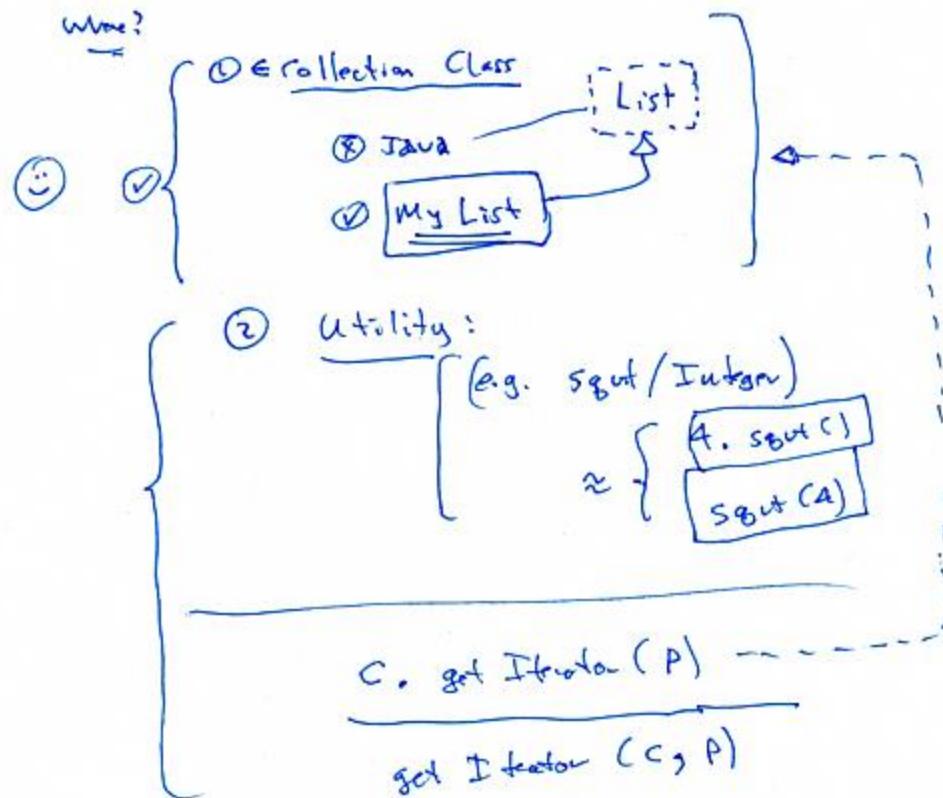


- Now we know what to do, how to implement it?

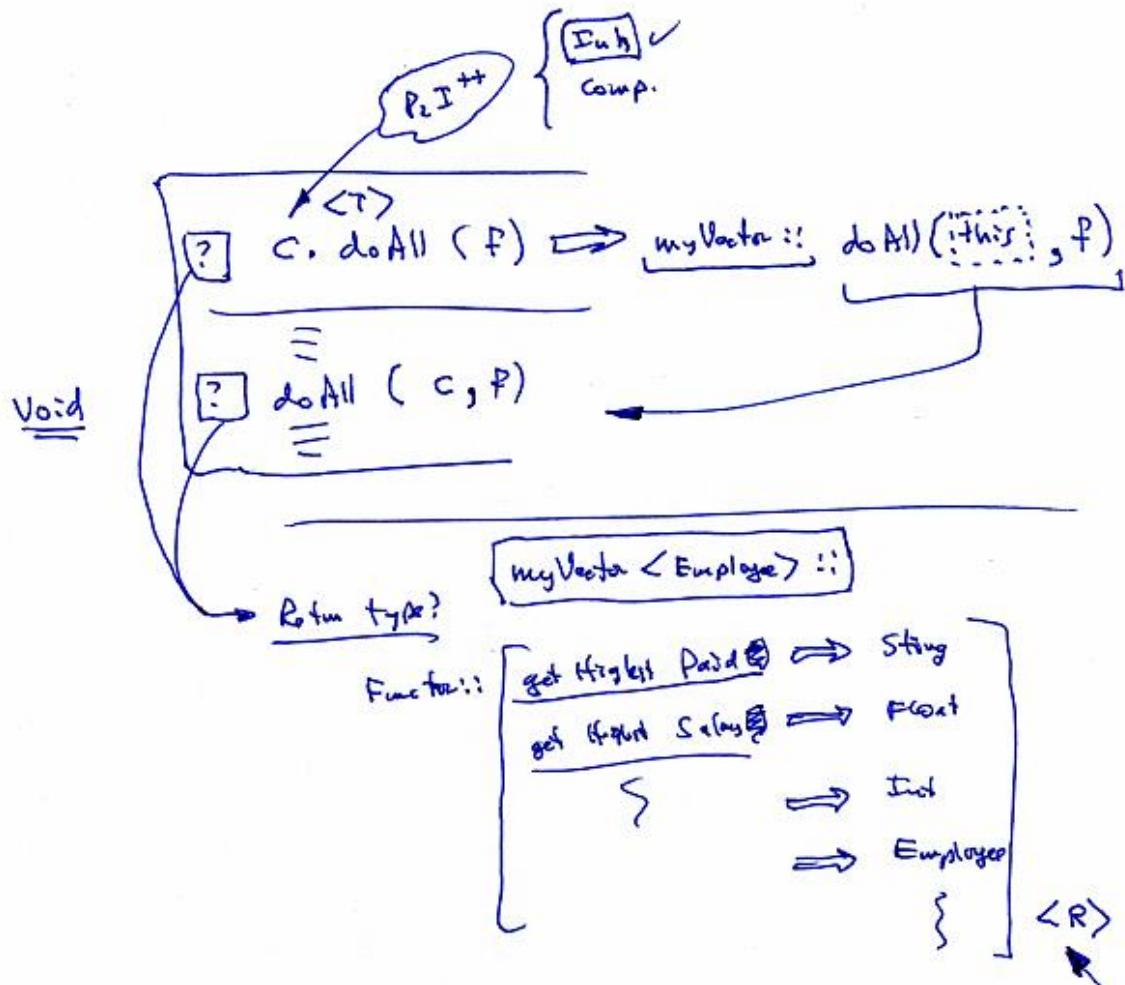


- There are multiple ways to do this;
 - One is to chain multiple iterators,
 - Another is to have a single filter with a compound predicate
 - Another would be to have one “*mega-predicate*” with multiple logic selections- in it, but that is a poor design...

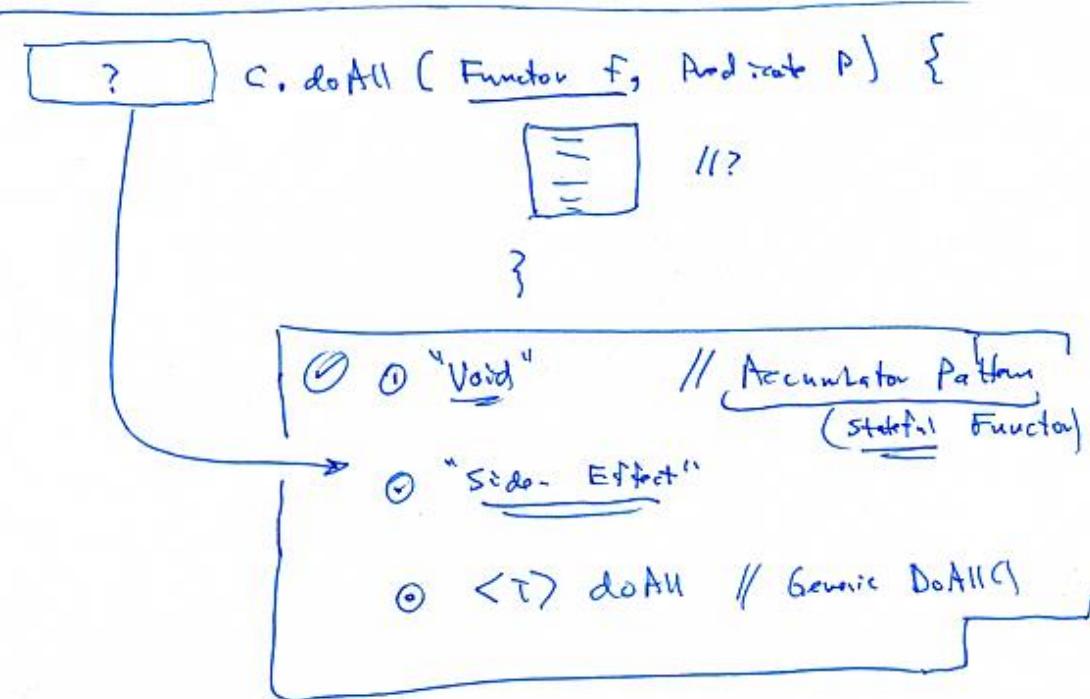
④ New (Smart) Iterators (Where?)



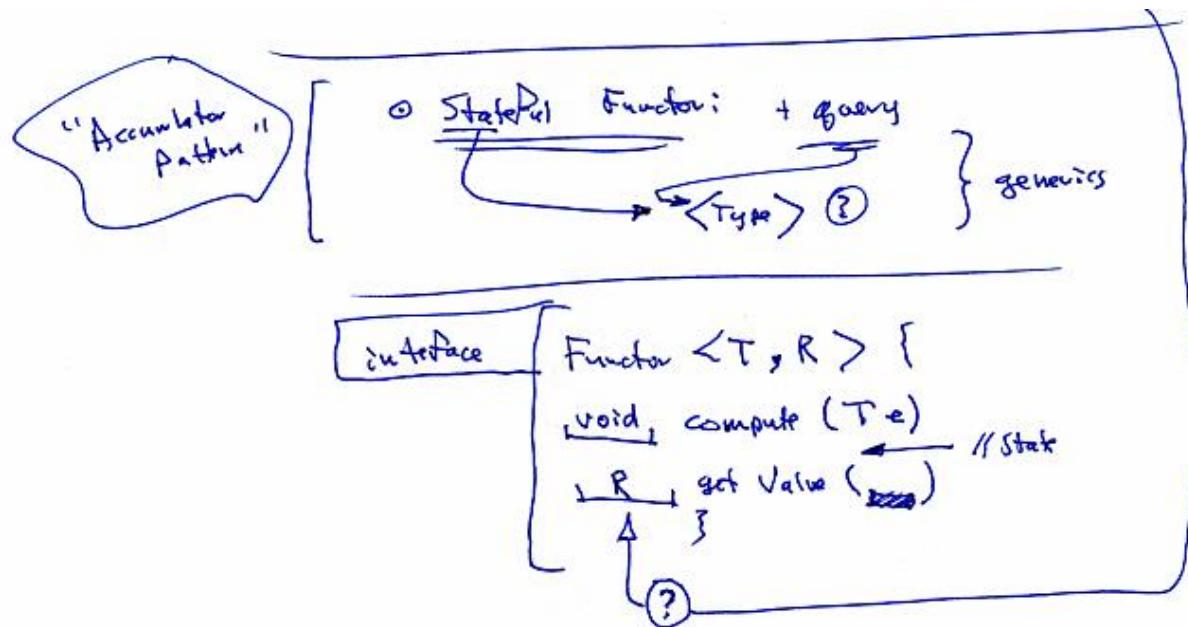
- Where should the new methods be implemented?
- In the collection class (we don't own it), or as separate utilities?



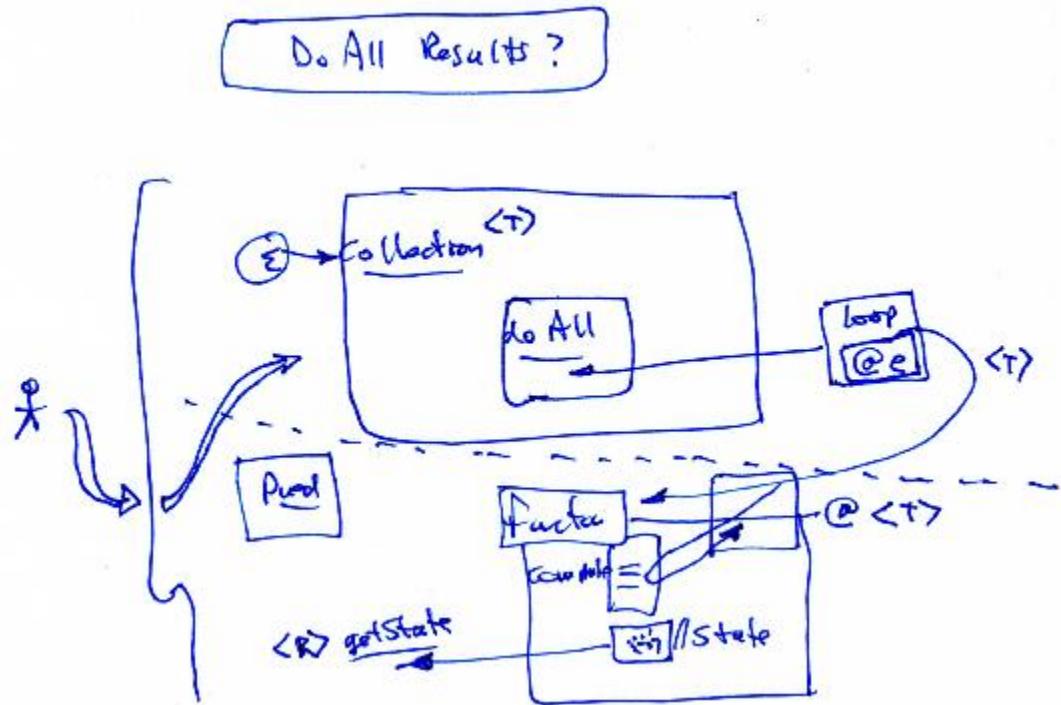
- Other design issues – return type of *doAll*?
- Functor is element-by-element
- Never sees context (= collection)
- *doAll()* uses it for a collection, how to return results?



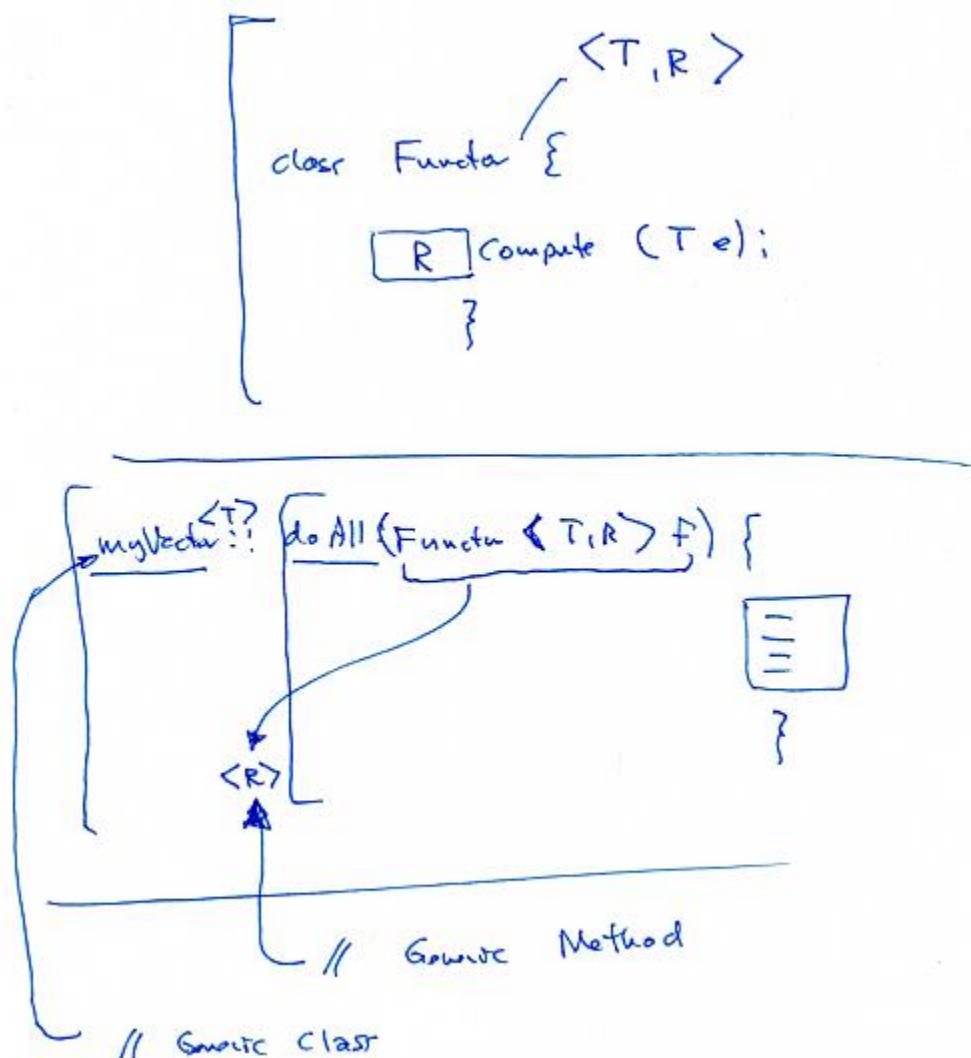
- three options:



- We will use the accumulator pattern, a stateful functor

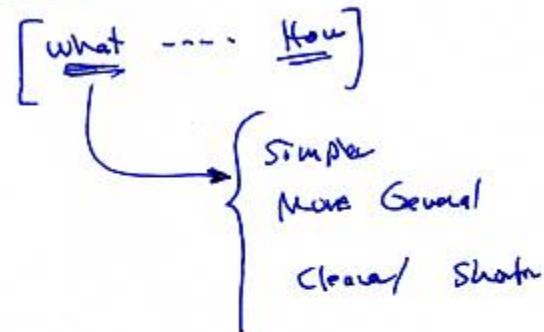


- `doAll` return via stateful functor



- An alternative would be a generic `doAll()` method, different return types each time, depending on the argument types!

① Declarative Style:



② SCI!!:

[Name :: Form]

... in Name

Describe ← Creates ← Is

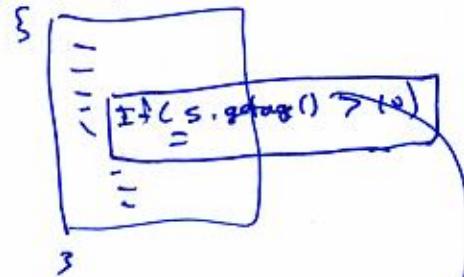
③ CS: Semantic Gap

- Higher level (functional) languages are more declarative
- The ultimate goal would be that (language \approx result), $name \approx form$ from SCI

Lab:::

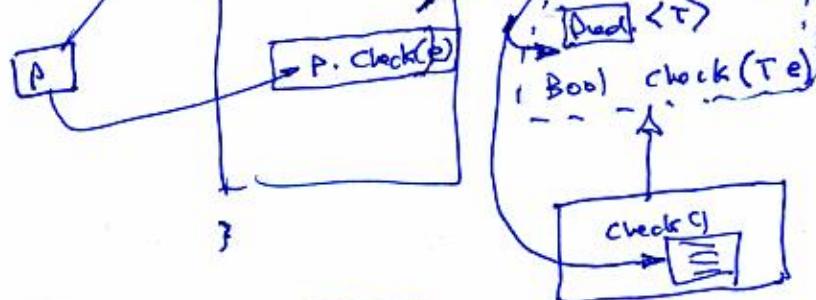
Iterator

get Swimmers By Age ()

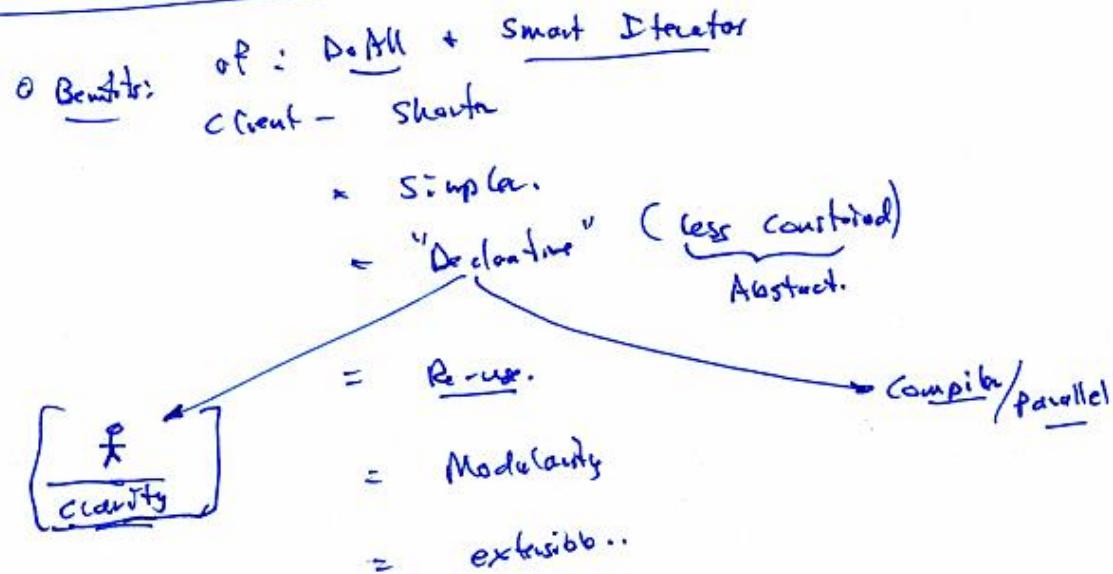


Iterator

getSwimmers ()



- The iterator lab; if you did it with a concrete specific filter with logic in the iterator,
It is easy to generalize and convert to use a predicate



- Benefits of *smart-iterators*

Day 6: Functional Iterators

Daily Topics

Day 6:**More Iterators...**

- Quiz & Review
- Iterator Exercise
 - Add internal iterator to *Vector* \Leftrightarrow *MyVector*
 - *accumulator* pattern
(or, *generic methods*)
- Iterators...
 - Selective iterator
 - make **Iterable** \rightarrow *views*
 - add compound predicates { composition... }
- FP style \rightarrow SE⁺⁺
 - Declarative; more abstract \Leftrightarrow simpler, shorter, clearer, ...
 - *Loopless* programming
- Standard FP functions (riders...)
 - Filter, map, reduce } List comprehensions | Introspection
 - Reduce \Leftrightarrow foldr, foldl, ...
- Exercise/Lab
 - add selective to *myVector*
 - then: selective & internal
- Lab:
 - Iterator Lab'
 - FP \rightsquigarrow IP
 - ... (JSQL, ~)
 - { Guava
 - } Scala
 - Contrasted Generics.

Day 7a: More Iterators & FP

Daily Topics

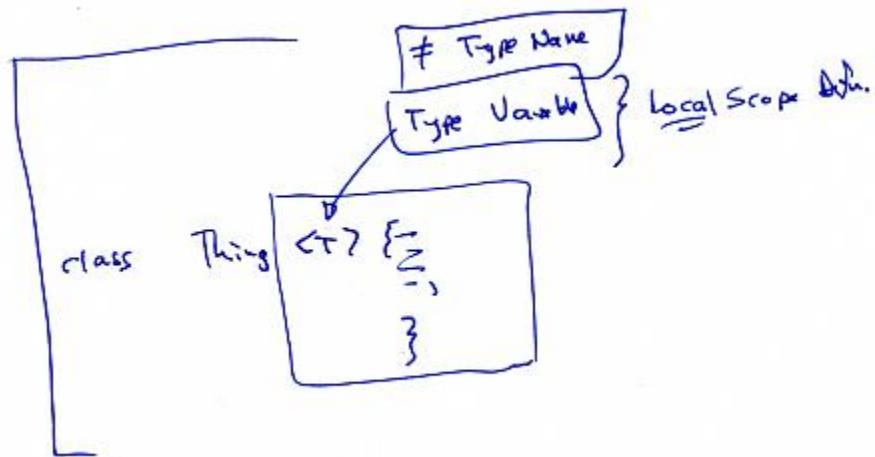
Day 6a:

Functional Iterators...

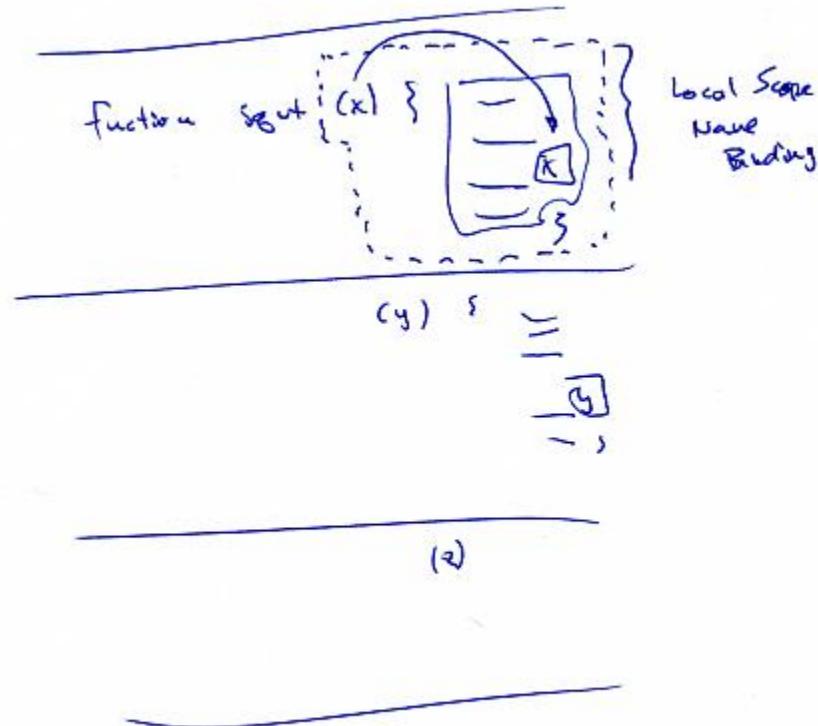
- Iterator Exercise Review
- Iterators...
 - make *Iterable* → *views*
 - add compound predicates
- FP style → **SE⁺⁺**
 - Declarative; more abstract ⇒ simpler, shorter, clearer, ...
 - *Loopless* programming \textcircled{O}
- Examples...
 - Guava iterators $\textcircled{[Guava]}$
- Generators

Lazified Iterators
Predicate Algebra (DSL)

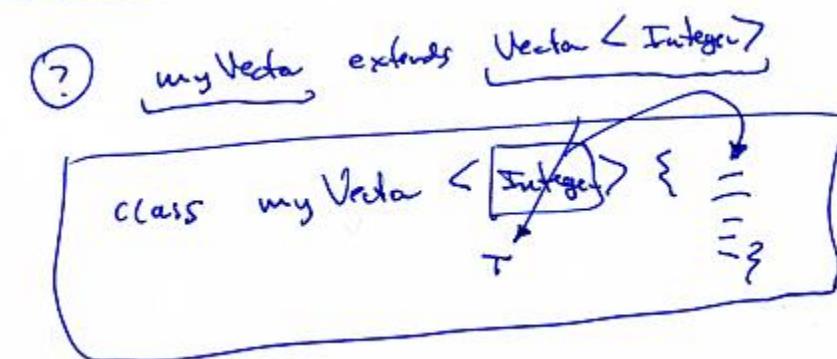
Review of Generics::



- Generics are type abstractions
- The scope of the type declaration (`<T>`) is the following body of the class



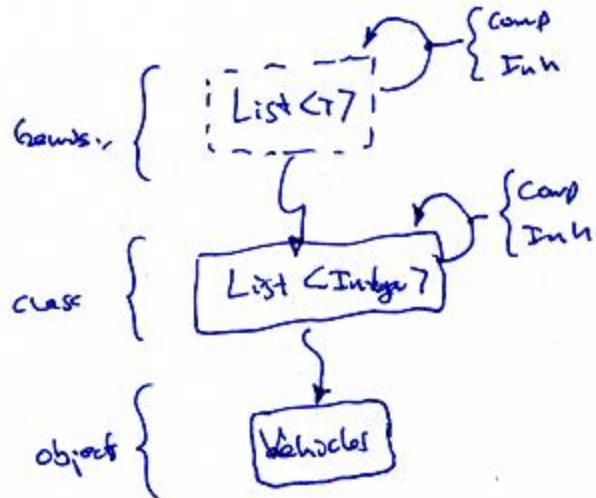
- Like all abstraction names, the actual name is unimportant (example of function parameters)



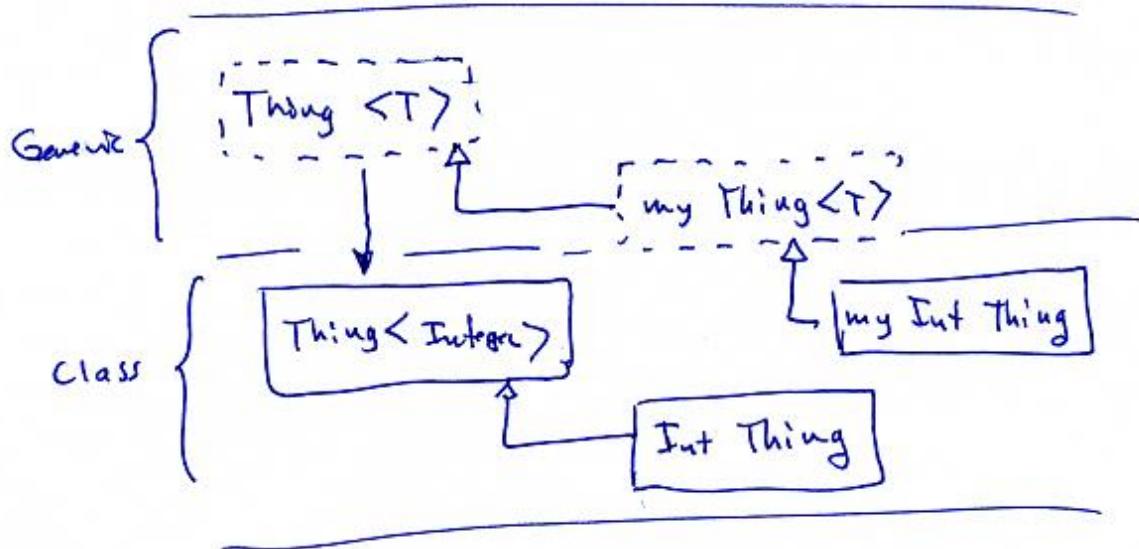
- Don't confuse type abstraction (parameter) names with concrete type names, the compiler won't! ☺

$\rightarrow (\neq \text{Raw}) \leftrightarrow \text{Error}$

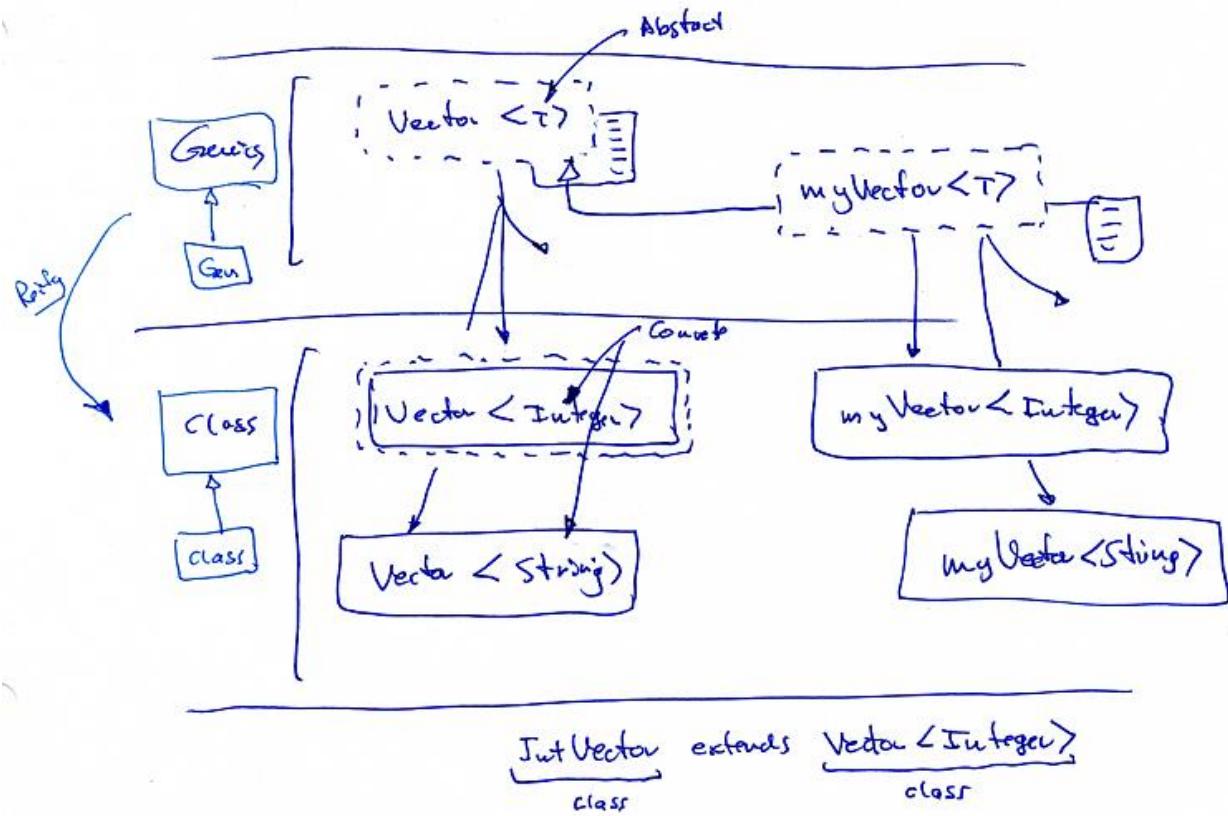
① Generics :: $\approx \text{meta-class}$



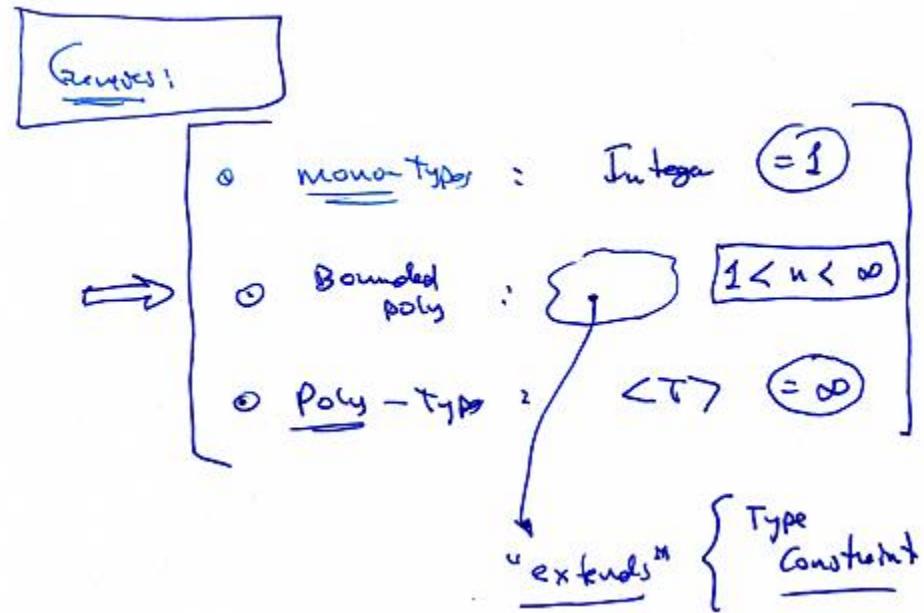
- Generics and Classes must be composed at their own levels



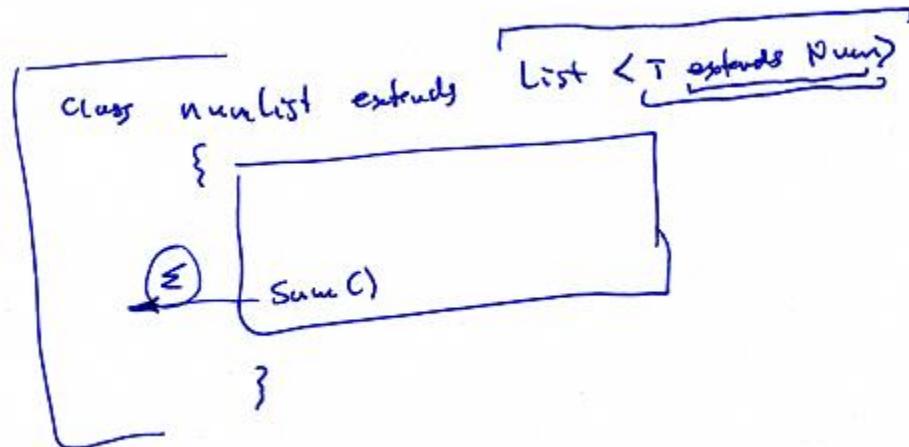
- Proper usage – instantiate a generic to a concrete, and then inherit



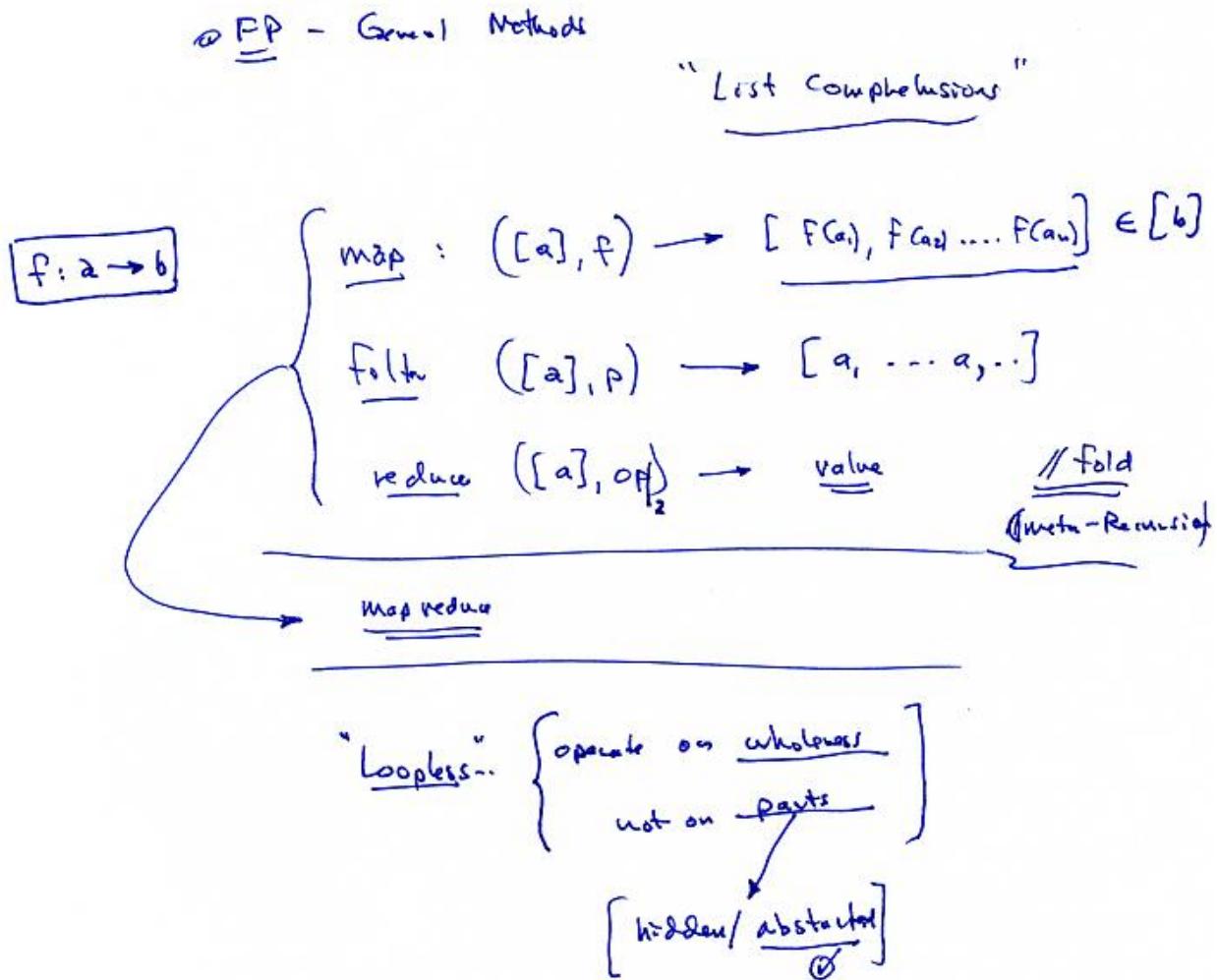
- Examples



- Bounded generics



- Bounded Generics allow methods to be type specific



- Functional programming basics

"Yoga tāk kuru Karmanī"

"Meditate & Then Act"

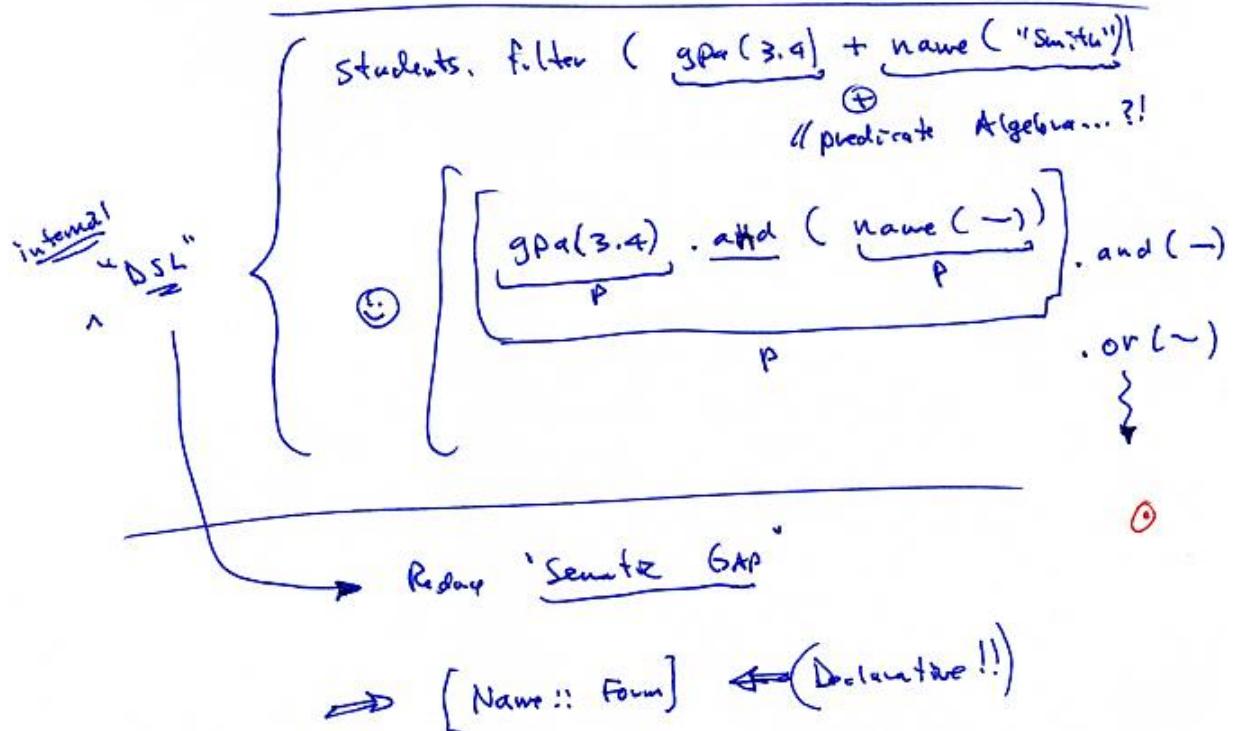
- Parallel SCI principle

Review of Iterator Exercise::

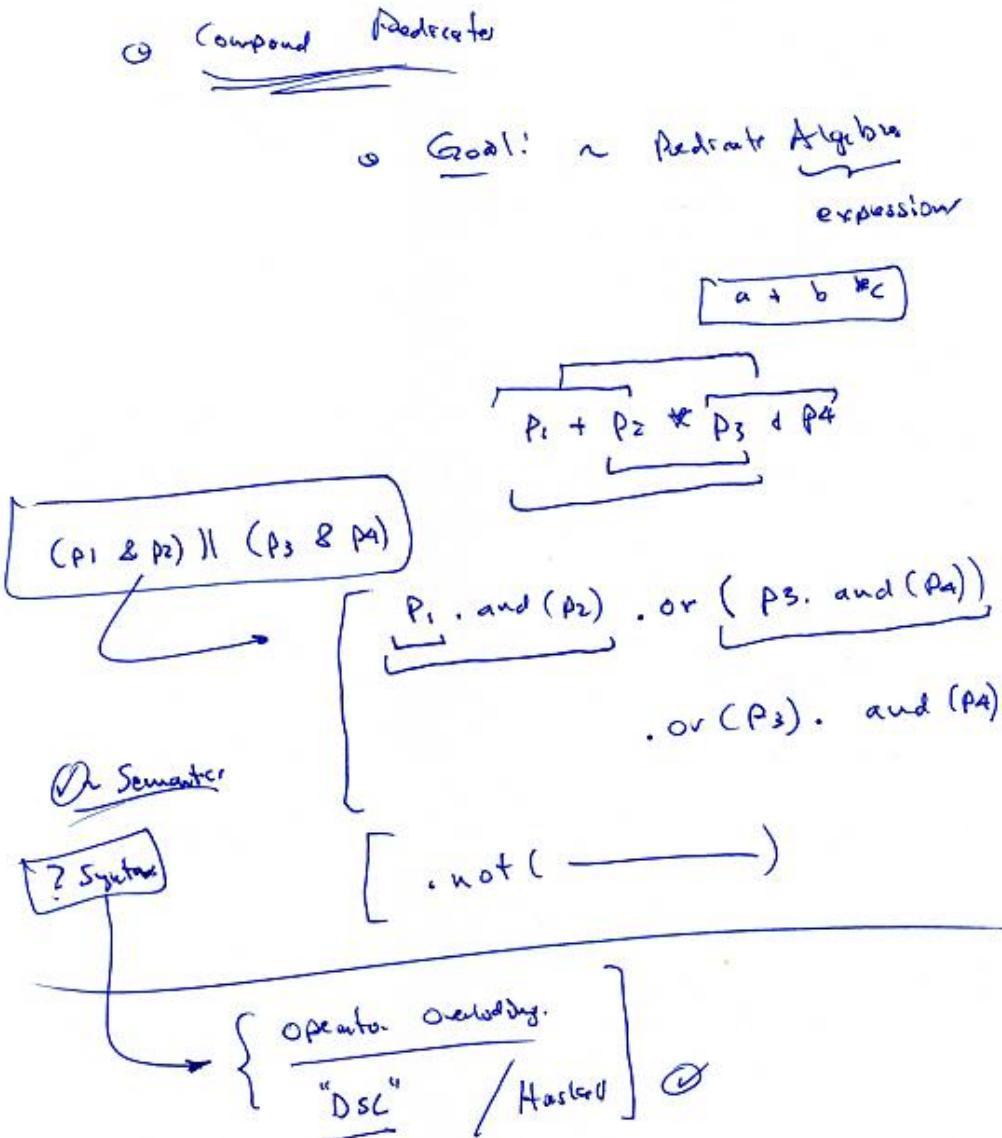
cs525: Pattern Notes

Iterator Styles:

		Design ✓	Adapter!	Arch Impl?
		Complete Traversal	Selective Traversal	
External	[<i>Iterator<T> it = c.iterator();</i> <i>while (it.hasNext())</i> <i>ProcFun(it.next())</i>] Or, [<i>foreach (T e : c) { ... }</i>]	[<i>Iterator<T> it = c.iterator(Pred p);</i> [Then use <i>normally!</i> ??]]		[<i>if (p(...))</i>]
	<i>c.doAll(ProcFun)</i>	<i>c.doAll(Predicate, ProcFun)</i>		[<i>if (p(...))</i>]



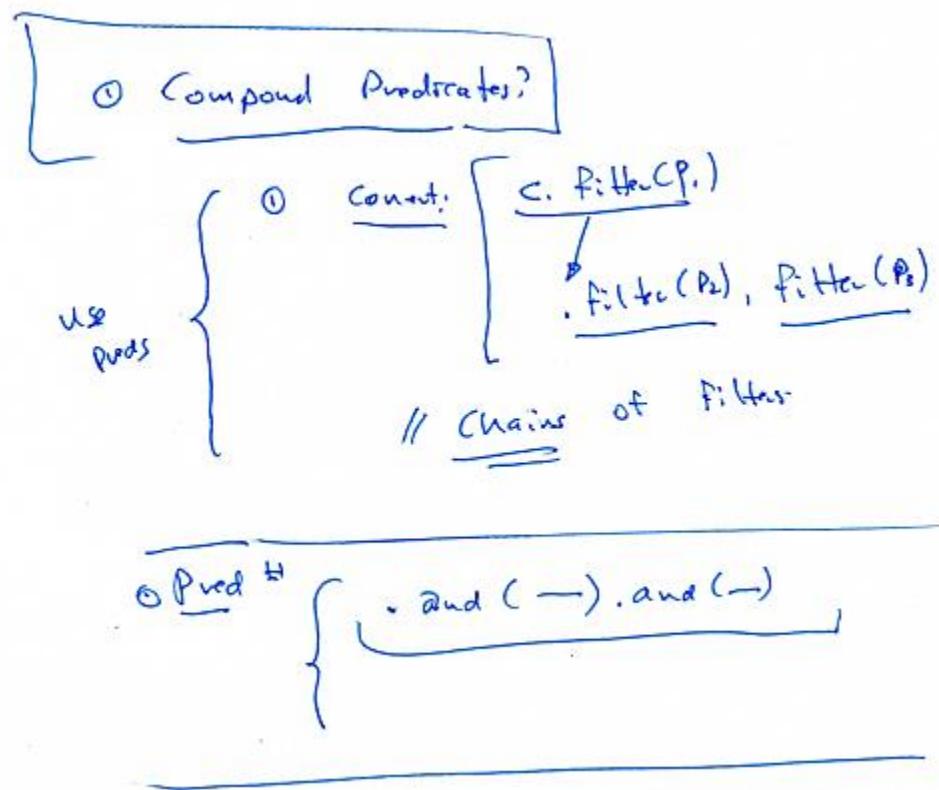
- Chaining of predicates



- Compound Predicates

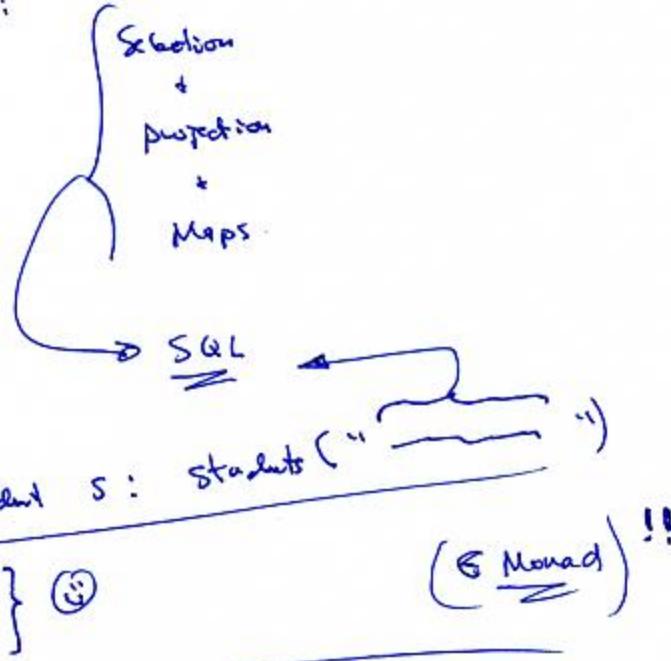
$\left\{ \begin{array}{c} \underline{\text{"Hello"} \quad ++ \quad \underline{\text{"World"}}} \\ \text{double } \boxed{\text{sqrt.}} \end{array} \right.$

- We know how to compose data structures, but no convenient way to do similar things (in Java) with functions.



- How to add compound predicates

④ JSQL:



⑤ "Linq" ≈ Java | Linq

- Extending the idea of predicates to general queries

 **guava-libraries**

Guava: Google Core Libraries for Java 1.5+

Project Home Downloads Wiki Issues Source

Search Current pages for Search

[Introduction](#)
[Basic Utilities](#)
[Collections](#)
[Caches](#)
Functional Idioms
[Obtaining](#)
[Using Predicates](#)
[Using Functions](#)
[Concurrency](#)
[Strings](#)
[Primitives](#)
[Ranges](#)
[I/O](#)
[Hashing](#)
[EventBus](#)
[Math](#)
[Releases](#)
[Tips](#)
[Mailing List](#)
[Stack Overflow](#)

« **FunctionalExplained**
Functional Idioms in Guava, explained.
 explained Update

Caveats

As of Java 7, functional programming in Java can only be approximated through awkward and verbose use of anonymous classes. This is expected to change in Java 8, but Guava is currently aimed at users of Java 5 and above.

Excessive use of Guava's functional programming idioms can lead to verbose, confusing, unreadable, and inefficient code. It's best to use them most easily (and most commonly) abused parts of Guava, and when you go to preposterous lengths to make your code look cool, your team weeps.

Compare the following code:

≈ Functor

```

Function<String, Integer> lengthFunction = new Function<String, Integer>() {
    public Integer apply(String string) {
        return string.length();
    }
};
Predicate<String> allCaps = new Predicate<String>() {
    public boolean apply(String string) {
        return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
    }
};
Multiset<Integer> lengths = HashMultiset.create(
    Iterables.transform(Iterables.filter(strings, allCaps), lengthFunction));
  
```

- Example of Google libraries, like ours! ☺

```

/*
 * Created on Apr 30, 2005
 *
 * Functional objects made even easier with tiger
 * Alex Winston's Blog
 * http://weblogs.java.net/blog/alexwinston/archive/2005/01/functional_obje1.html
 *
 * for flexibility, would you want to do <? super T> on both
 * the Predicate and Closure within each method signature?
 * I believe this would then allow:
 *   foreach(AccountImpl[], Predicate<Account>, Closure<Account>)
 *
 * GRG: Convert to Java Collections
 */
package functional;

// import org.apache.commons.collections.*;
import java.util.*;

public class Preposition {
    public static <T> void with(T subject, Functor<T> p) {
        p.execute(subject);
    }

    public static <T> void when(T subject, Predicate<T> p, Functor<T> c) {
        if (p.evaluate(subject))
            Preposition.with(subject, c);
    }

    public static <T> void foreach(T[] subjects, Functor<T> c) { // do All
        for (T subject : subjects)
            Preposition.with(subject, c);
    }

    public static <T> void foreach(T[] subjects, Predicate<T> p, Functor<T> c) {
        for (T subject : subjects)
            Preposition.when(subject, p, c);
    }

    public static <T> void foreach(Collection<T> subjects, Functor<T> c) {
        for (T subject : subjects)
            Preposition.with(subject, c);
    }

    public static <T> void foreach(Collection<T> subjects, Predicate<T> p, Functor<T> c) {
        for (T subject : subjects)
            Preposition.when(subject, p, c);
    }
}

```

DSL

- Another example of libraries in the same style as ours

```

public class TestPreposition extends TestCase {
    public void TestPreposition() {
        Predicate<Student> isPassing = new Predicate<Student>() {
            public boolean evaluate(Student s) {
                return ((s.getGrade() == Grade.A || s.getGrade() == Grade.B));
            }
        };
        Predicate<Student> isNotAbsent = new Predicate<Student>() {
            public boolean evaluate(Student s) {
                return s.getDaysAbsent() == 0;
            }
        };
        final List<Student> honorRoll = new ArrayList<Student>();
        Functor<Student> addToHonorRoll = new Functor<Student>() {
            public void execute(Student student) {
                honorRoll.add(student);
            }
        };
        // Some test data...
        Student student = new Student(Grade.D, 0);
        when(student, isNotAbsent, addToHonorRoll);
        assertEquals(1, honorRoll.size());
        assertEquals(0, honorRoll.get(0).getDaysAbsent());
        honorRoll.clear();

        List<Student> students = Arrays.asList(new Student[] {
            new Student(Grade.A, 1),
            new Student(Grade.B, 0),
            new Student(Grade.C, 0) });
        foreach(students, isPassing, addToHonorRoll);
        assertEquals(2, honorRoll.size());
        assertEquals(Grade.A, honorRoll.get(0).getGrade());
        assertEquals(Grade.B, honorRoll.get(1).getGrade());
        honorRoll.clear();
    }
}

```

- And the application usage...

- Ruby: Ruby uses blocks for iterators

```

Collection.each { |name| print name }

3.upto(6) { |i| print i }

5.times { print "*" }

sum = 0
[ 1, 2, 3, 4] .each { |i| sum += i }

total=0
Employees.each { |e| total += e.salary }      // Total all salaries

Employees.find { |e| e.name=="Smith" }    // Find an Employee

Employees.collect { |e| e.salary *= 1.50 } // Give everyone a raise!

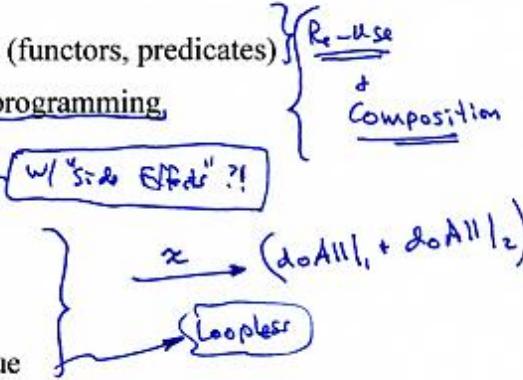
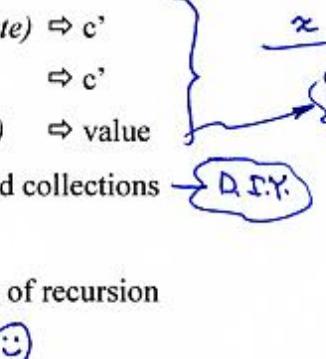
aButton = Button.new("Exit") { System.exit }      // actionListener!

```

- Ruby collections all provide methods which take blocks (closures) as an argument
 \Rightarrow functional arguments \approx FP

- Modern languages like Ruby already have all of this...
- Summary of Iterator Exercise

Functional Programming:

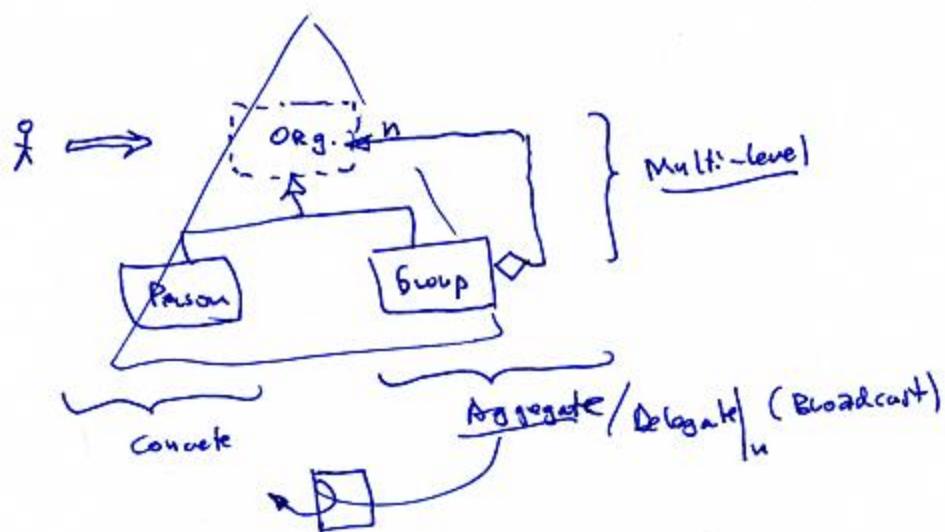
- An alternate programming paradigm
 - Dealing with wholeness, not parts
 - High level specification
 - ⇒ not *over-constrained*
 - ⇒ *parallelism, simplicity*
- Example: Internal & selective iterations
 - Are based on functional abstractions (functors, predicates)
 - Move in the direction of functional programming
 - *doAll(functor<T, R>)* 
- Three standard functional methods
 - filter: $c.filter(predicate) \Rightarrow c'$
 - map: $c.map(func) \Rightarrow c'$
 - reduce: $c.reduce(f_2, u) \Rightarrow \text{value}$ 
- All can be easily added to standard collections 
- Reduce $\equiv fold \{ foldr, foldl \}$
 - *Fold* \approx universal encoding of recursion
 - *Loopless* programming 
- And combining & parallelizing,
 - $c.mapReduce(f_m, f_r, u)$ 
- Exercise:
 - Define $\{ filter, map, doAll \}$ in terms of *reduce*...

Day 7b: Composite Pattern

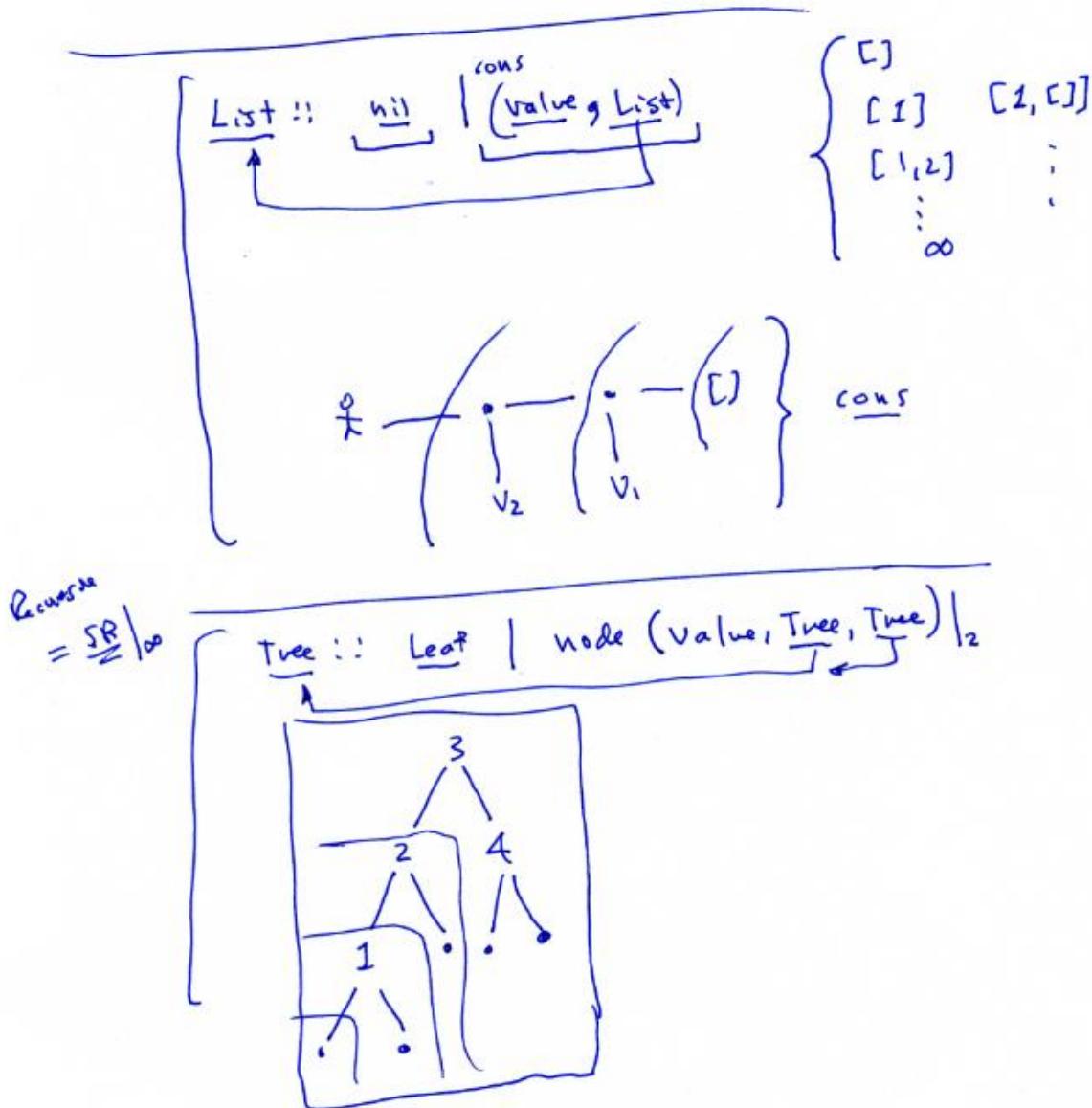
Daily Topics

Day 7b: Composite Pattern

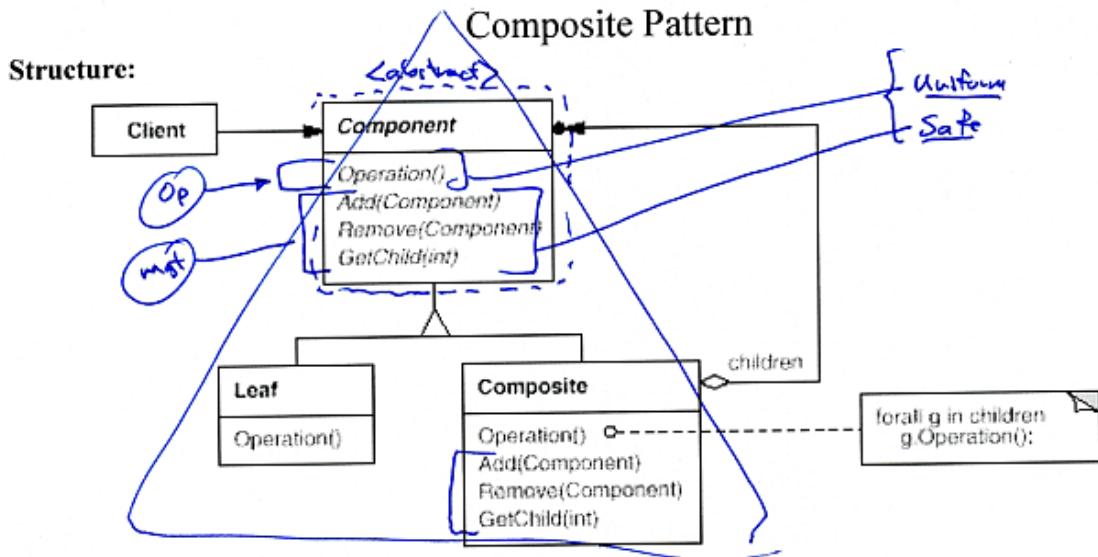
- Quiz & Review
- General representation of recursive data-structures
 - list, tree
- ⇒ Composite pattern
 - parts ≈ whole
 - based on SR loop ⇒ ∞ composition
- Design options
 - uniform, safe GOF
- Recursive iteration?
 - internal 2
 - external ?? ⊕
- Lab:
 - Composite Lab



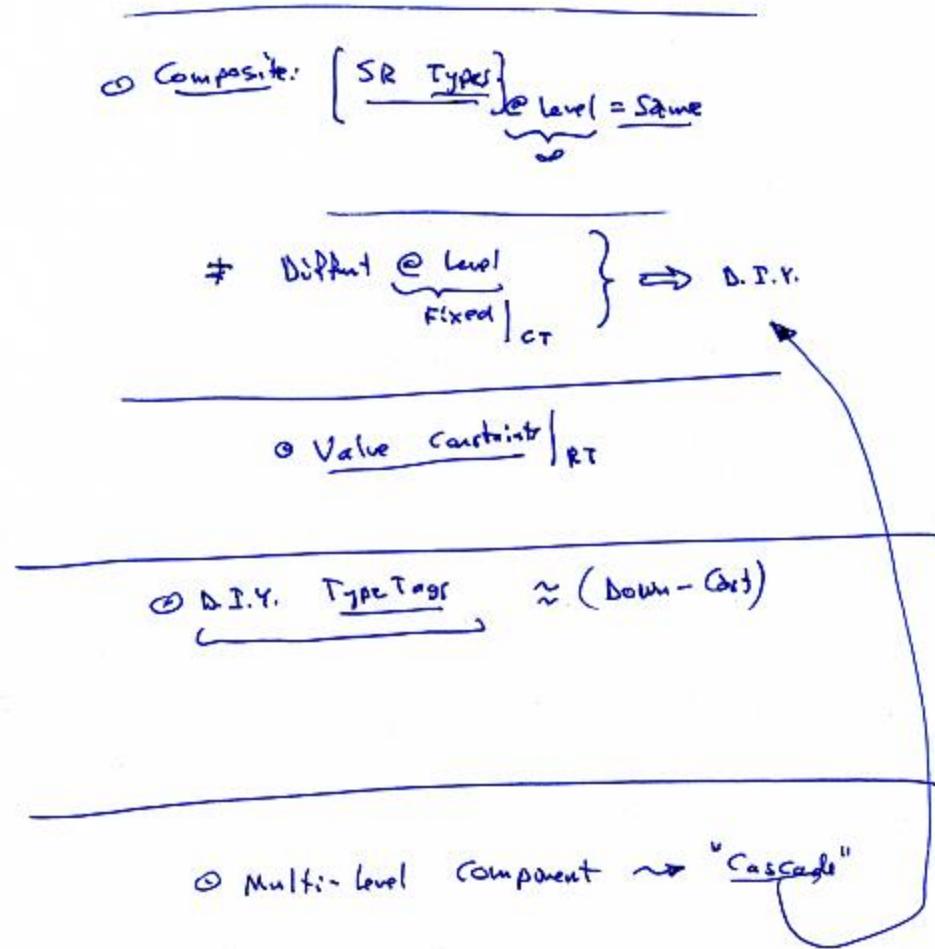
- General idea of Composite
- Recursive types, unified parts * & Whole



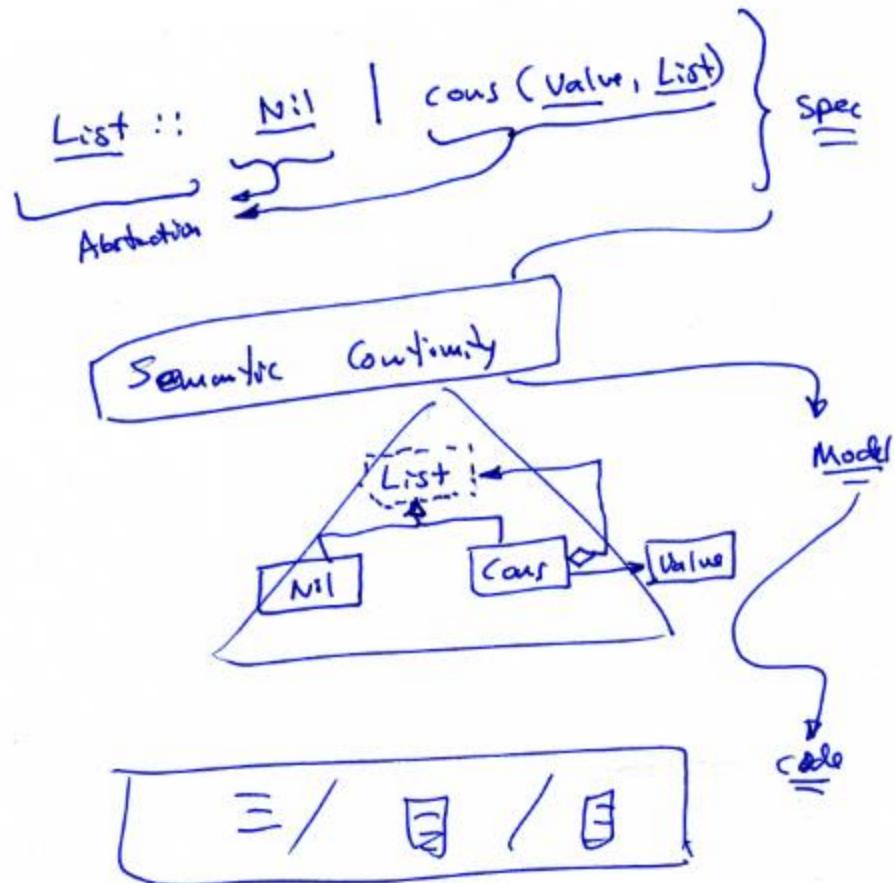
- Formal model of recursive types



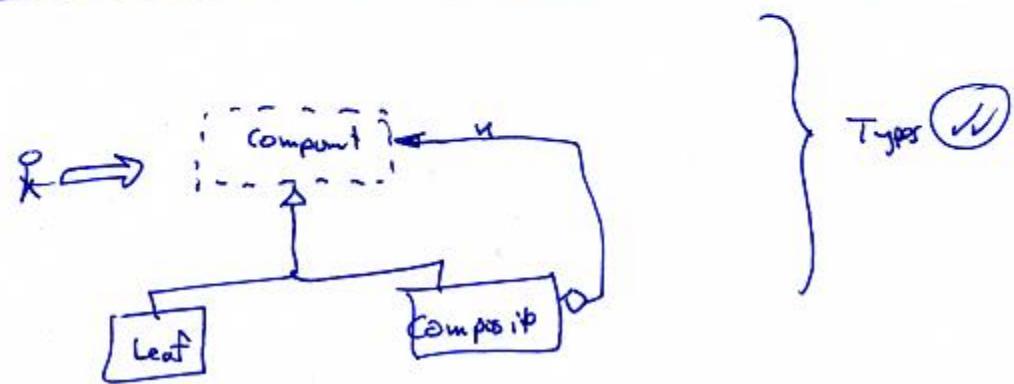
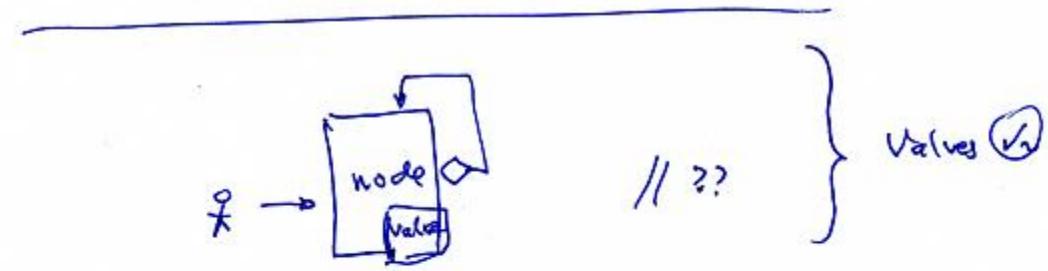
- GOF description of Composite Pattern



- Composite models systems with the same types at every level



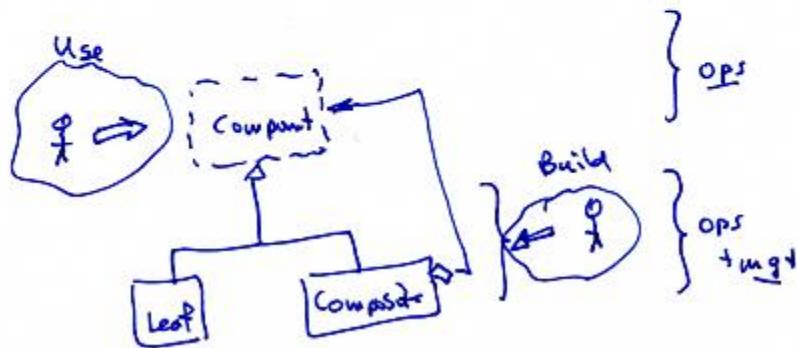
- Composite can be derived as an OO model of an abstract definition



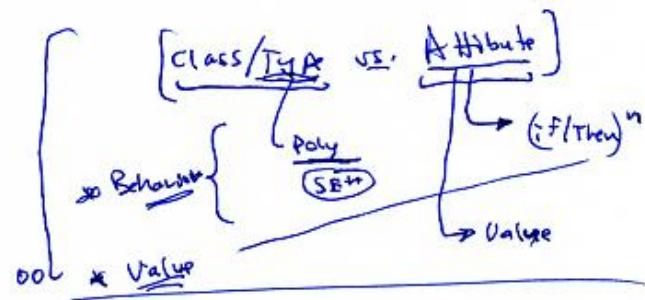
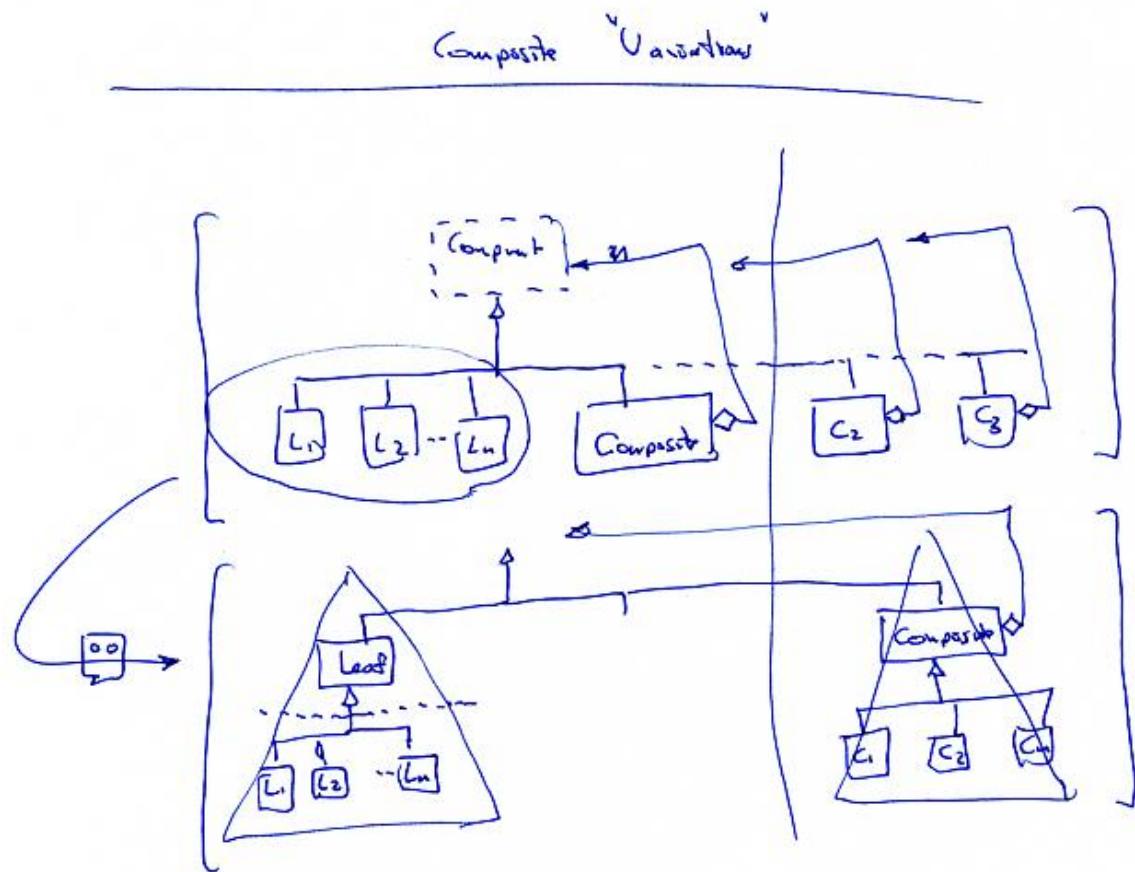
- Composite models the business (application) domain

① Composite

- o Build : (Sub-Type) \Rightarrow Set (Imgs.) $\xrightarrow{3}$ Abstract.
 - o Use : (Super/Parent) \Rightarrow Transf. (Ops.)
-



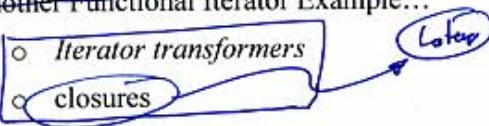
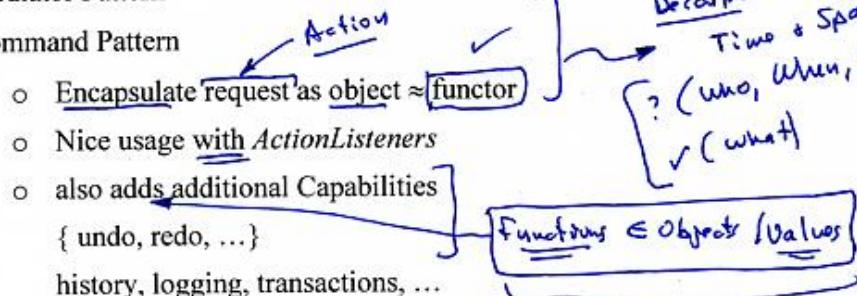
- Composite merges safe & Transparent
- Divides between management & operations messages

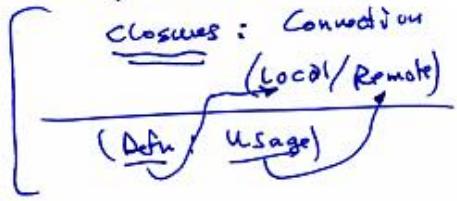


- Variations on Composite Pattern

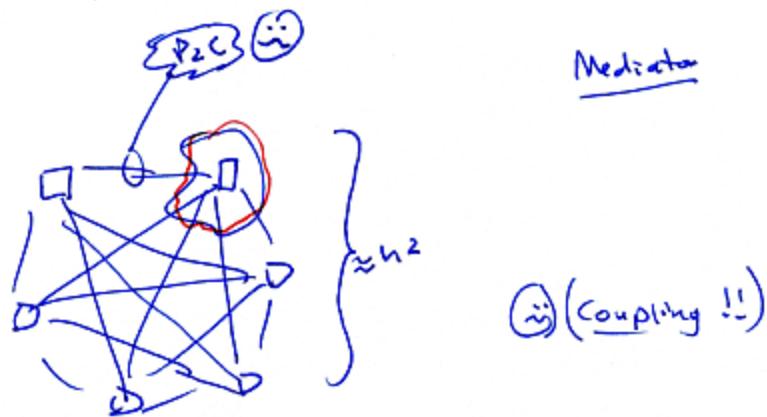
Day 8: Command & Mediator Patterns

Day 8: Command & Mediator Patterns

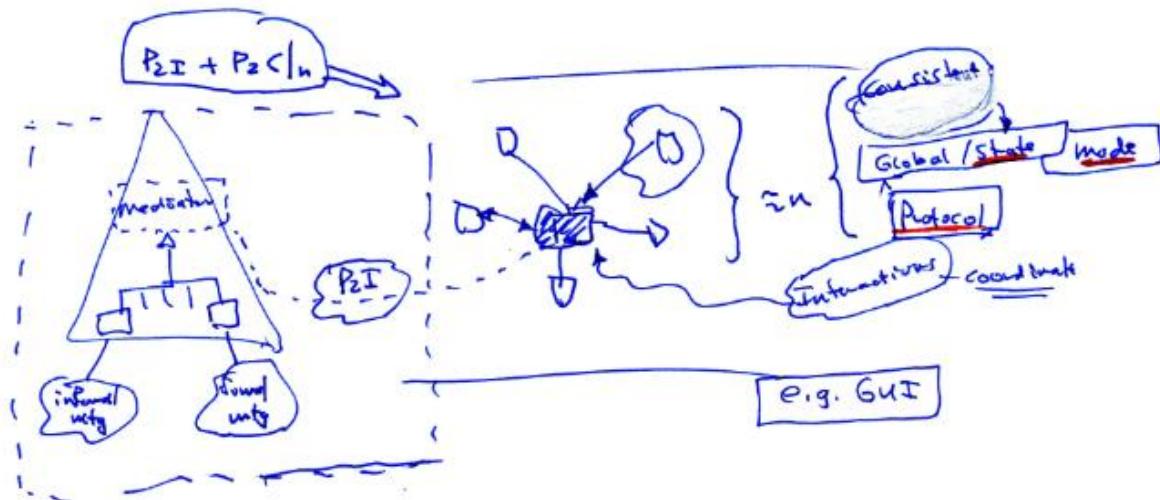
- Quiz & Review
- Another Functional Iterator Example...
 - *Iterator transformers*
 - *closures*
- Mediator Pattern
- Command Pattern
 - Encapsulate request as object ≈ functor
 - Nice usage with ActionListeners
 - also adds additional Capabilities
 - { undo, redo, ... }
 - history, logging, transactions, ...

Decouple:
Time + Space.
? (who, when, where)
✓ (what)
- Lab:
 - Command Lab

Closures: Connection
(Local / Remote)
(Defn) (Usage)

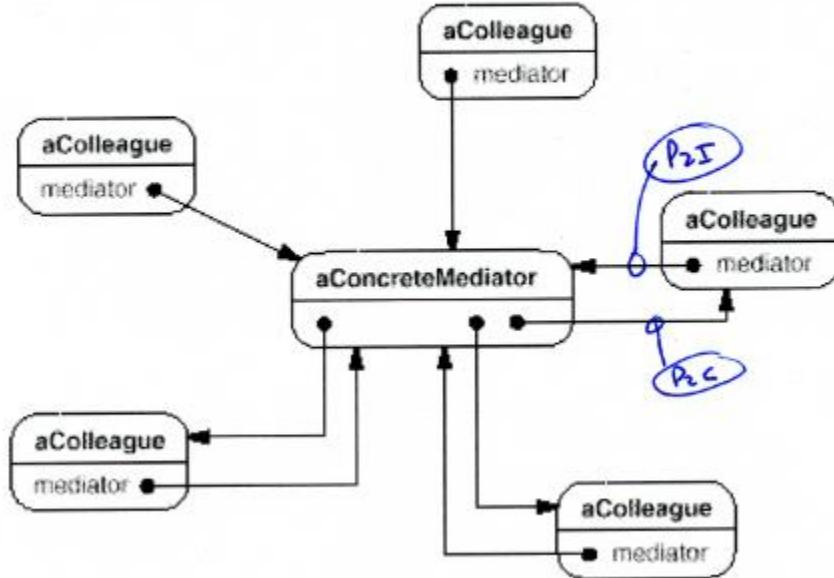


- Mediator helps resolve coupling in a sub-system

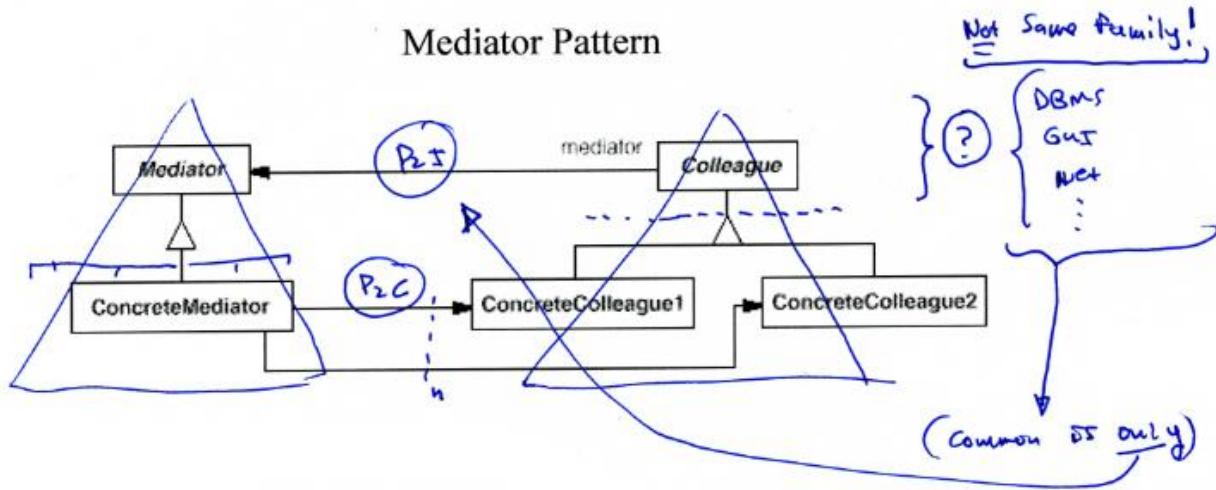


- Mediator centralizes protocol, coupling, and state (dynamically)

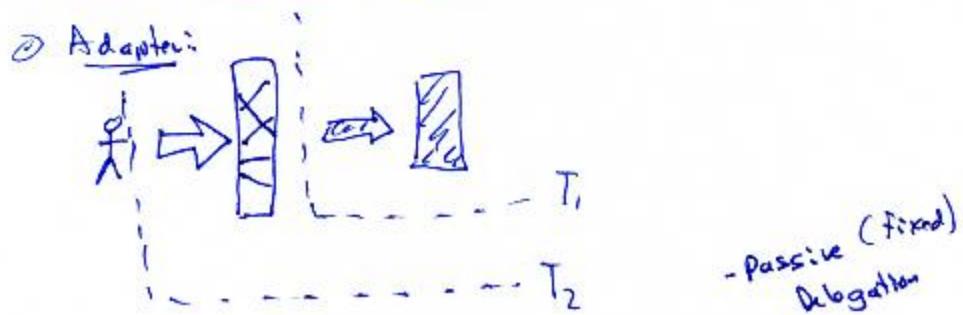
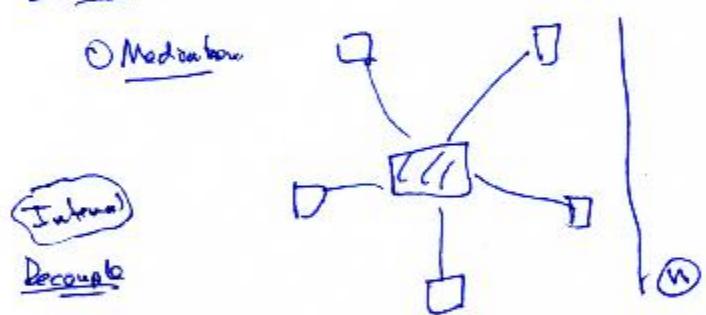
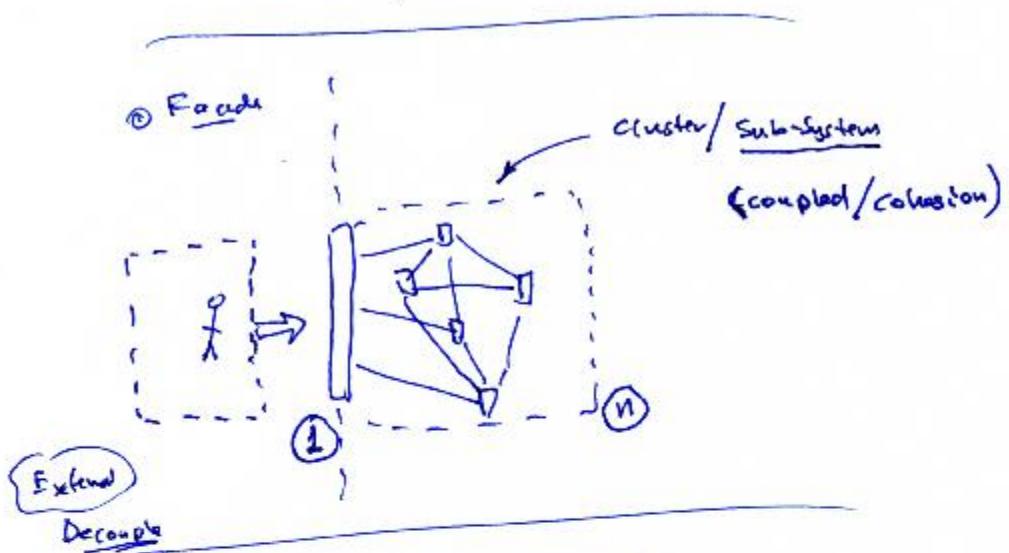
A typical object structure might look like this:



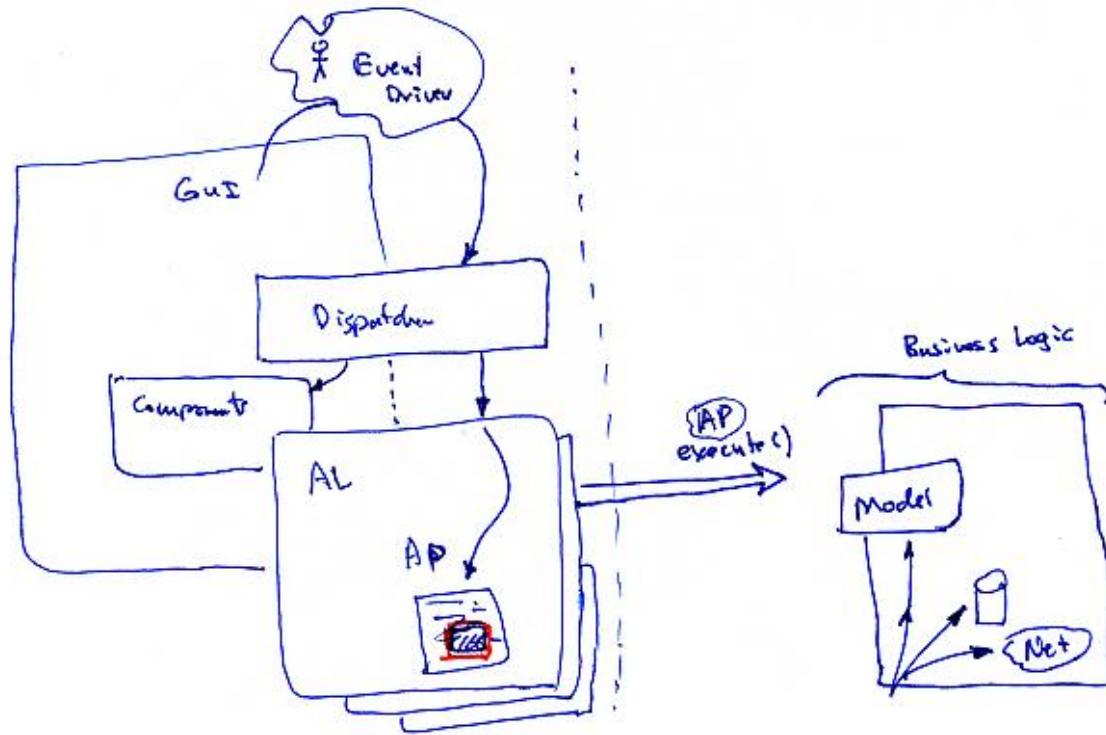
- GOF example



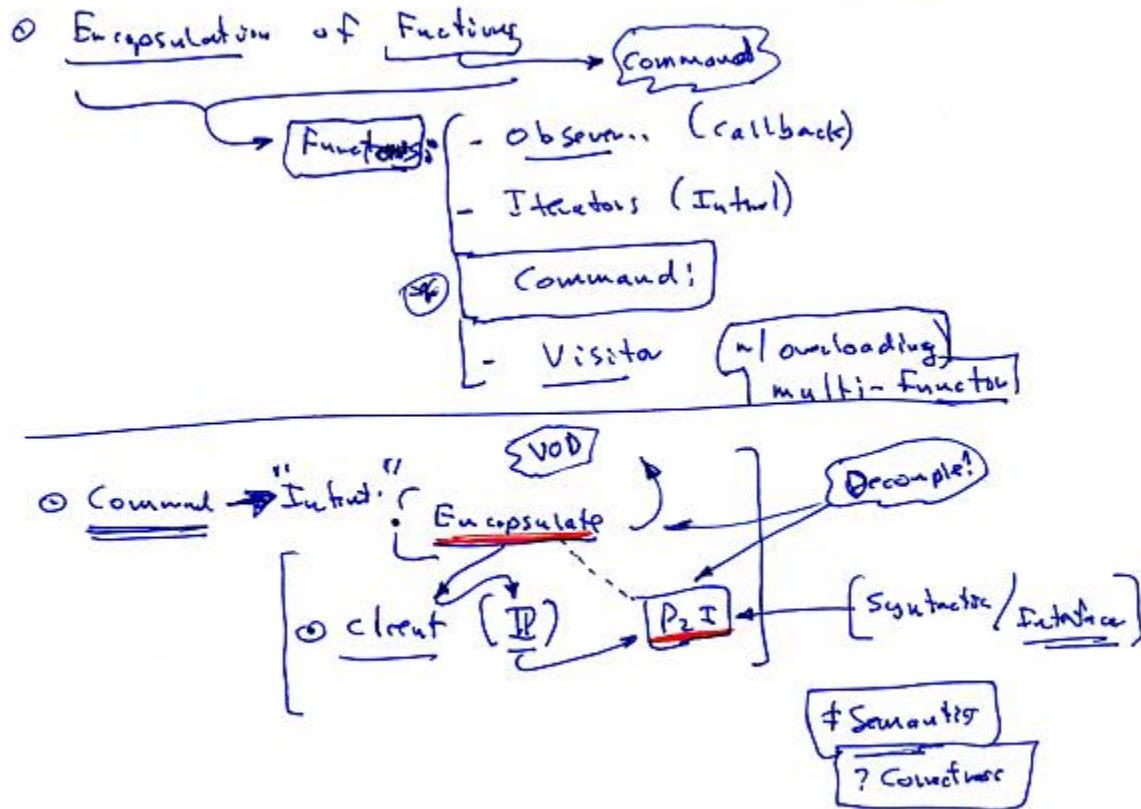
- GOF mediator



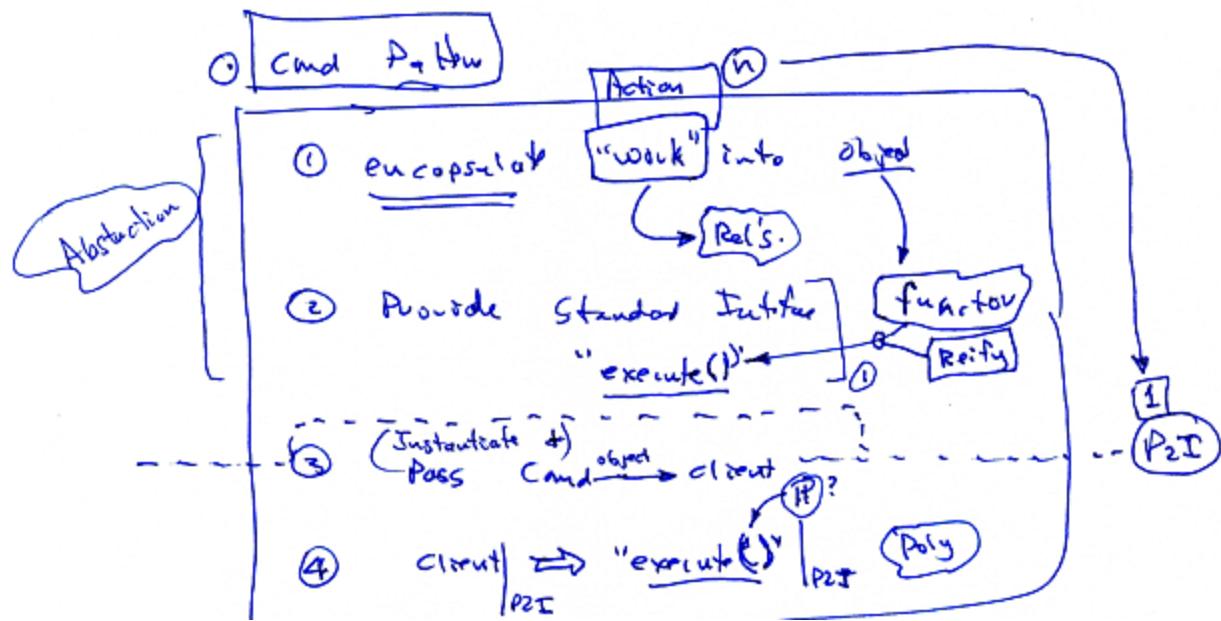
- Comparing Mediator and some other patterns

Command Pattern::

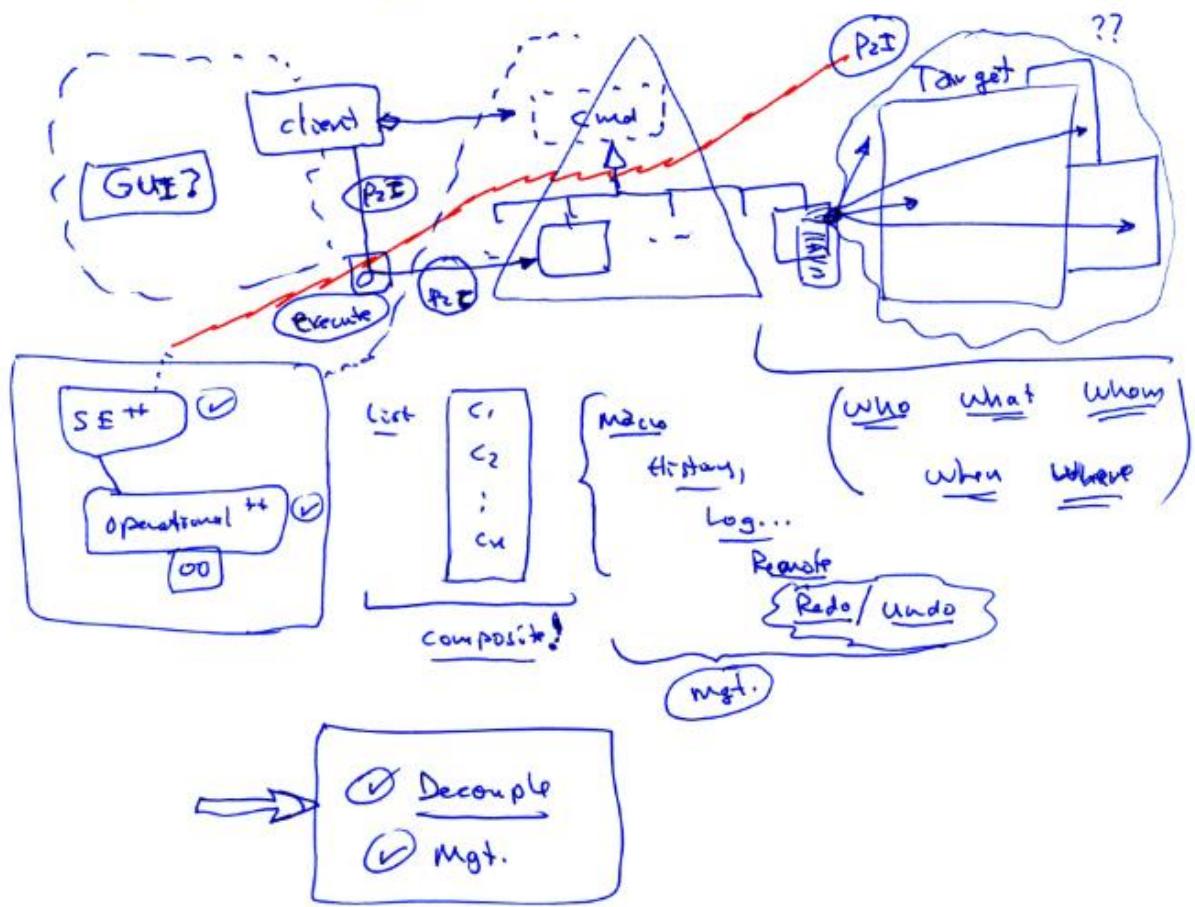
- The original Observer pattern
- AP ≈ Command?



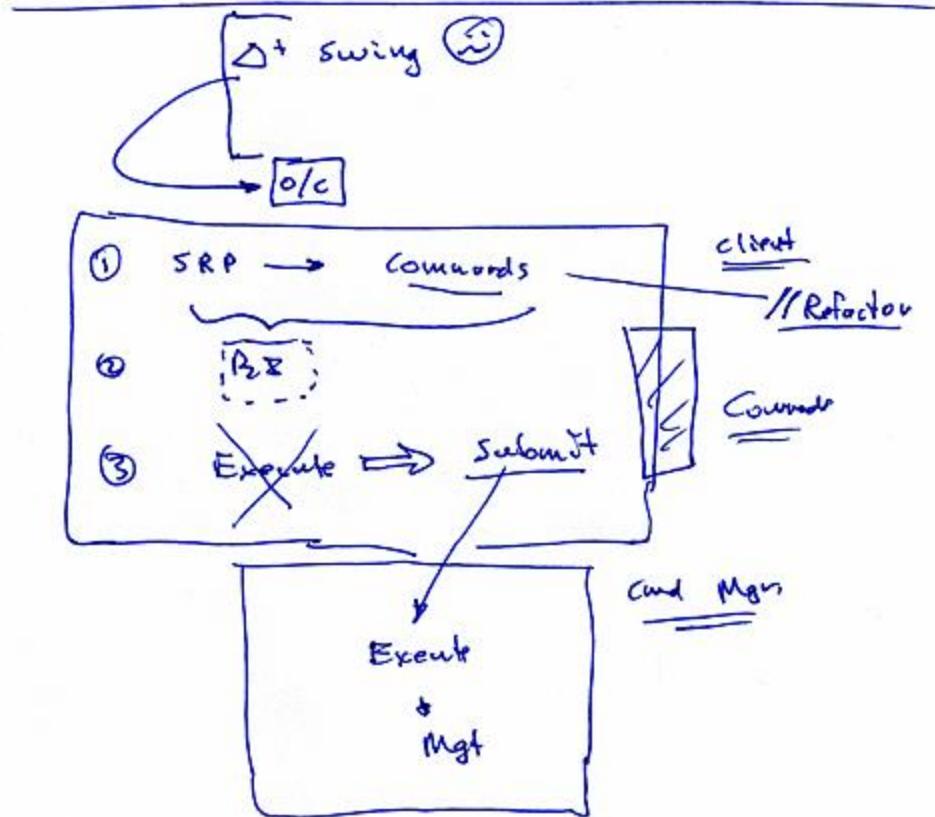
- Basic ideas of Command Pattern



- Steps of usage of Command pattern

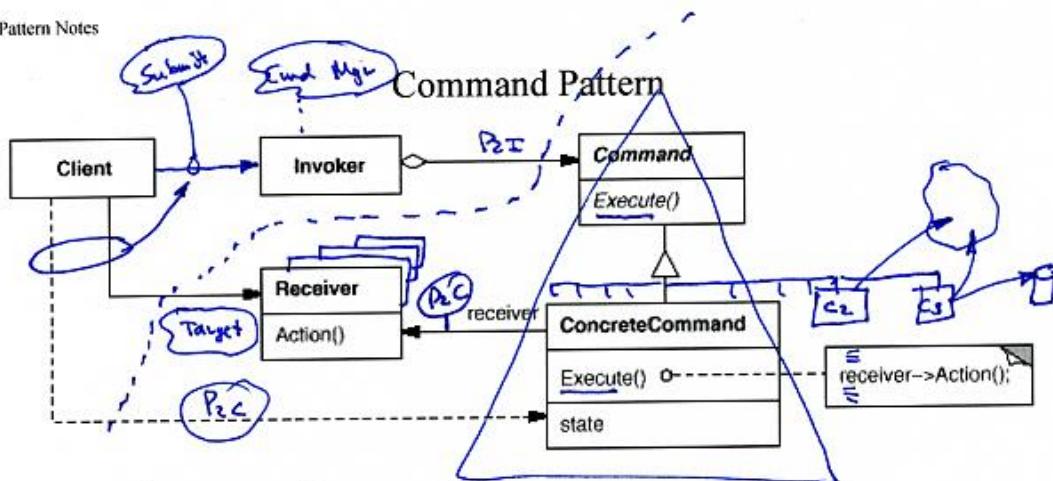


- Overall Architecture and usage of Command Pattern

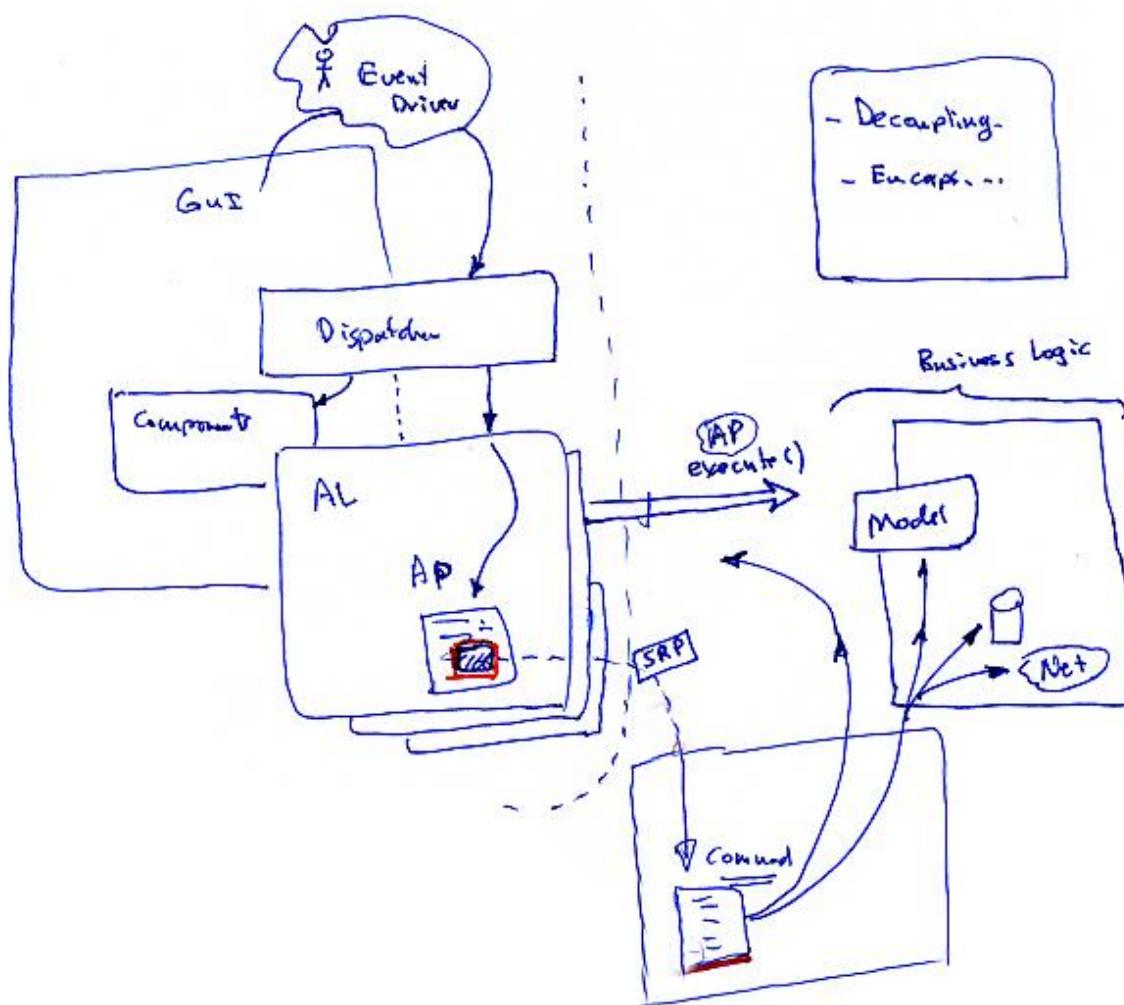


- Our lab uses GUI + Command pattern

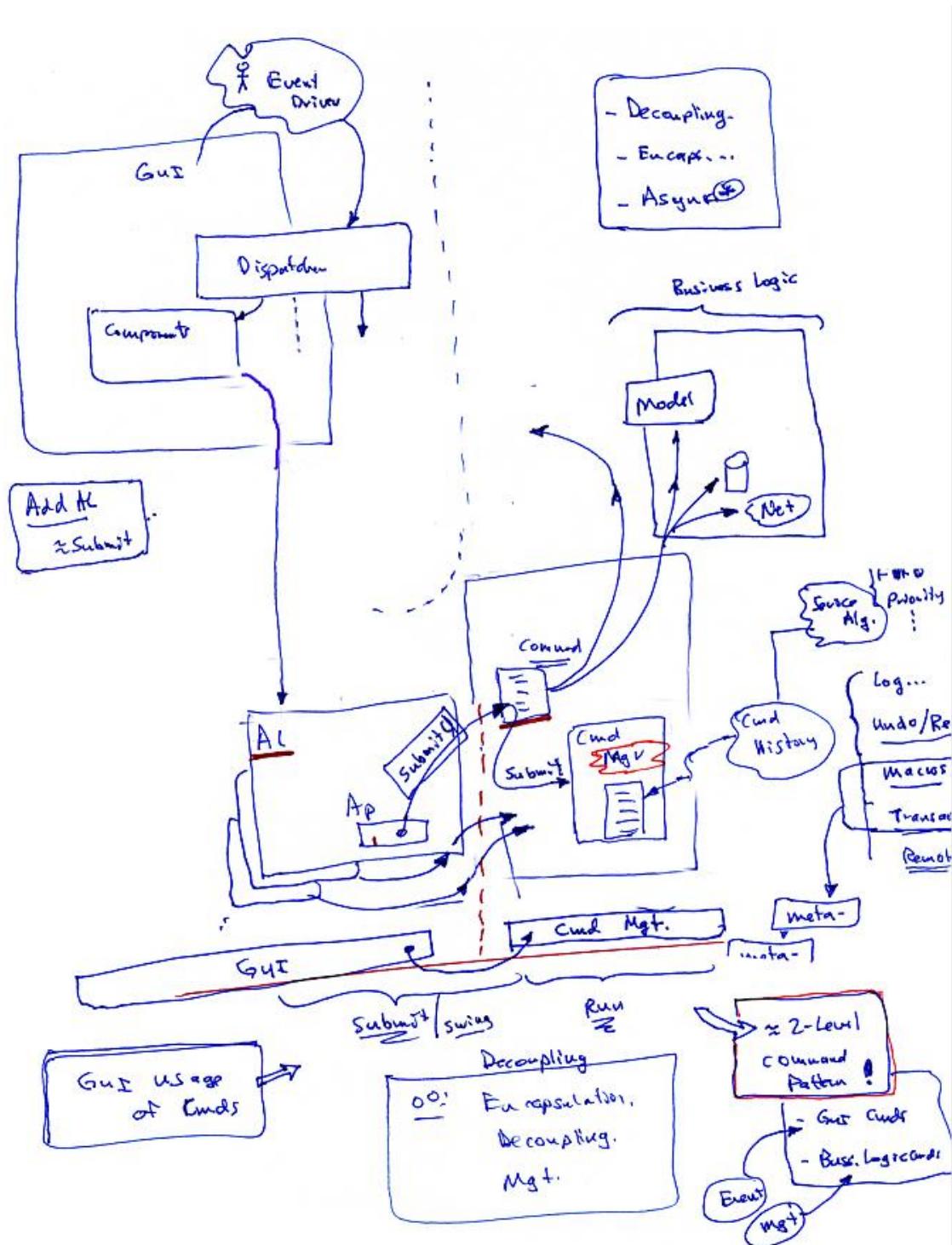
cs525: Pattern Notes



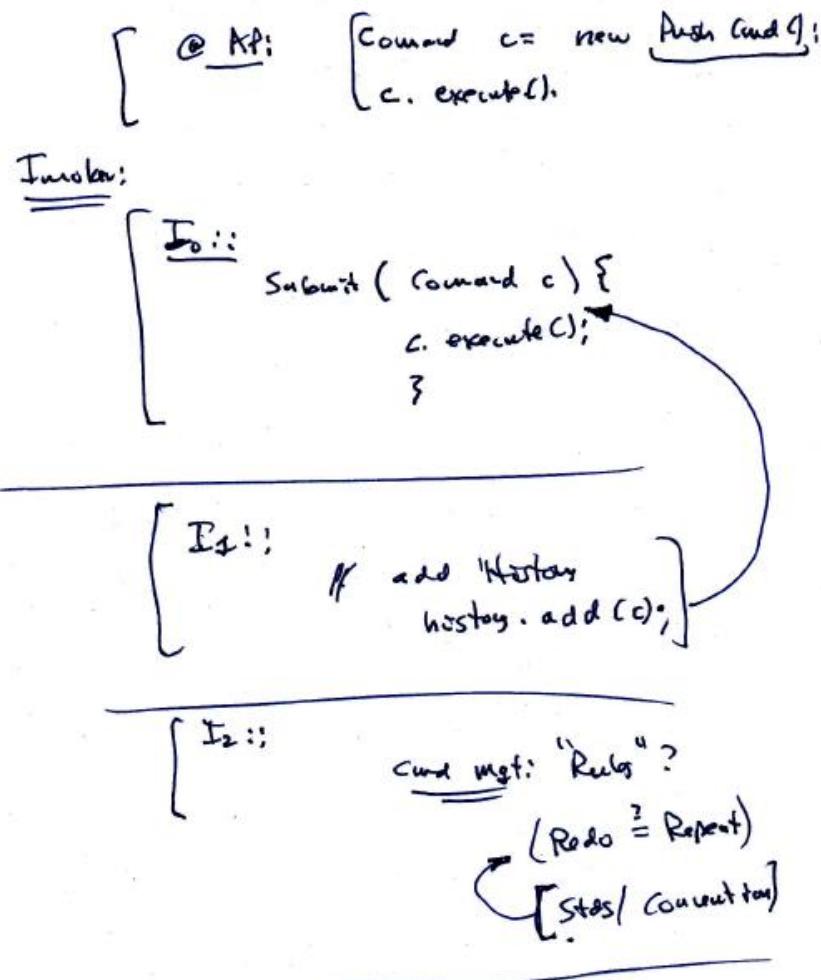
- GOF structure for Command Pattern



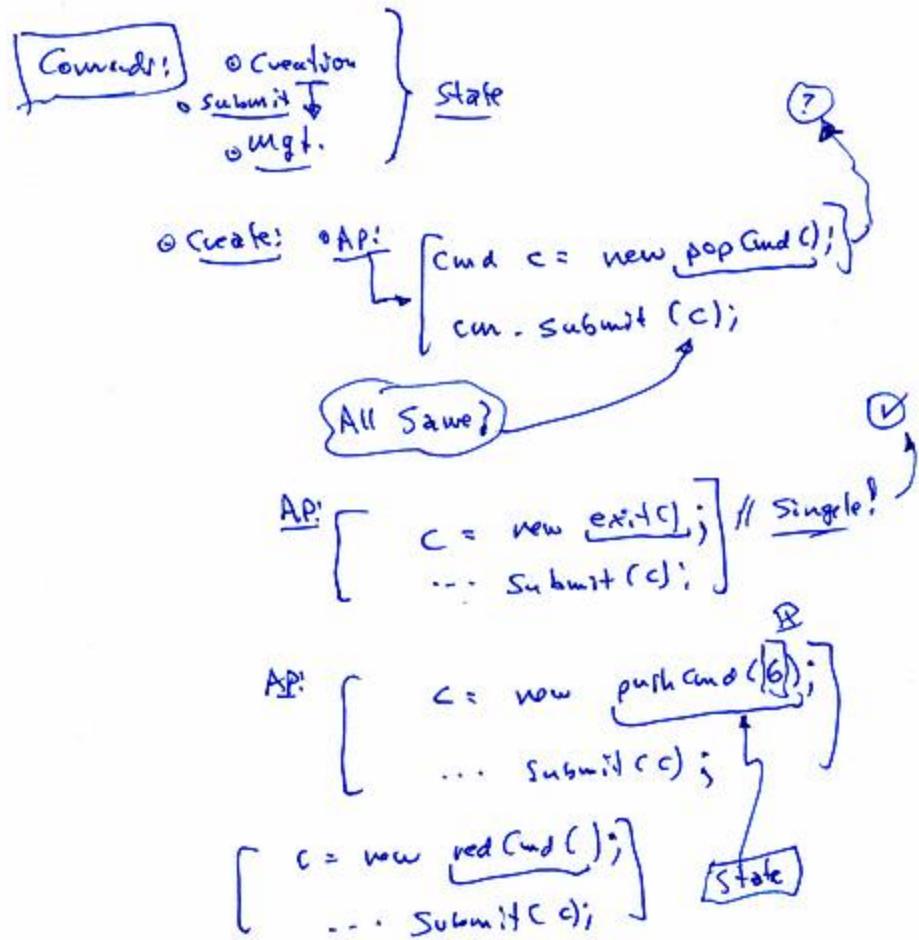
- Modify to separate AP from AL, Business logic from GUI



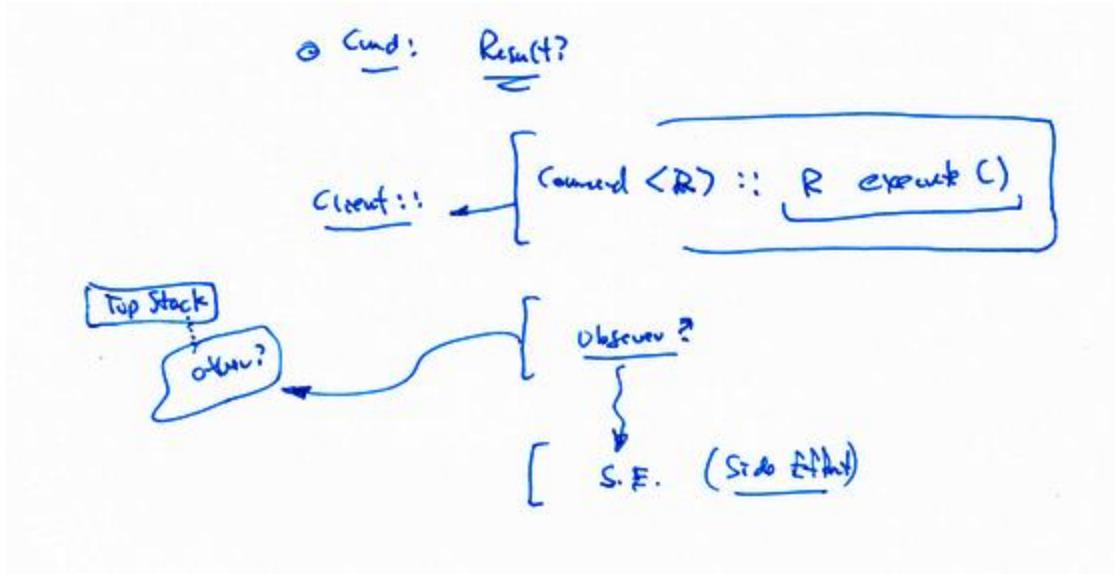
- Rewrite to use Command pattern
- Keep GUI dispatching using Observers



- Steps in using command Pattern in the lab

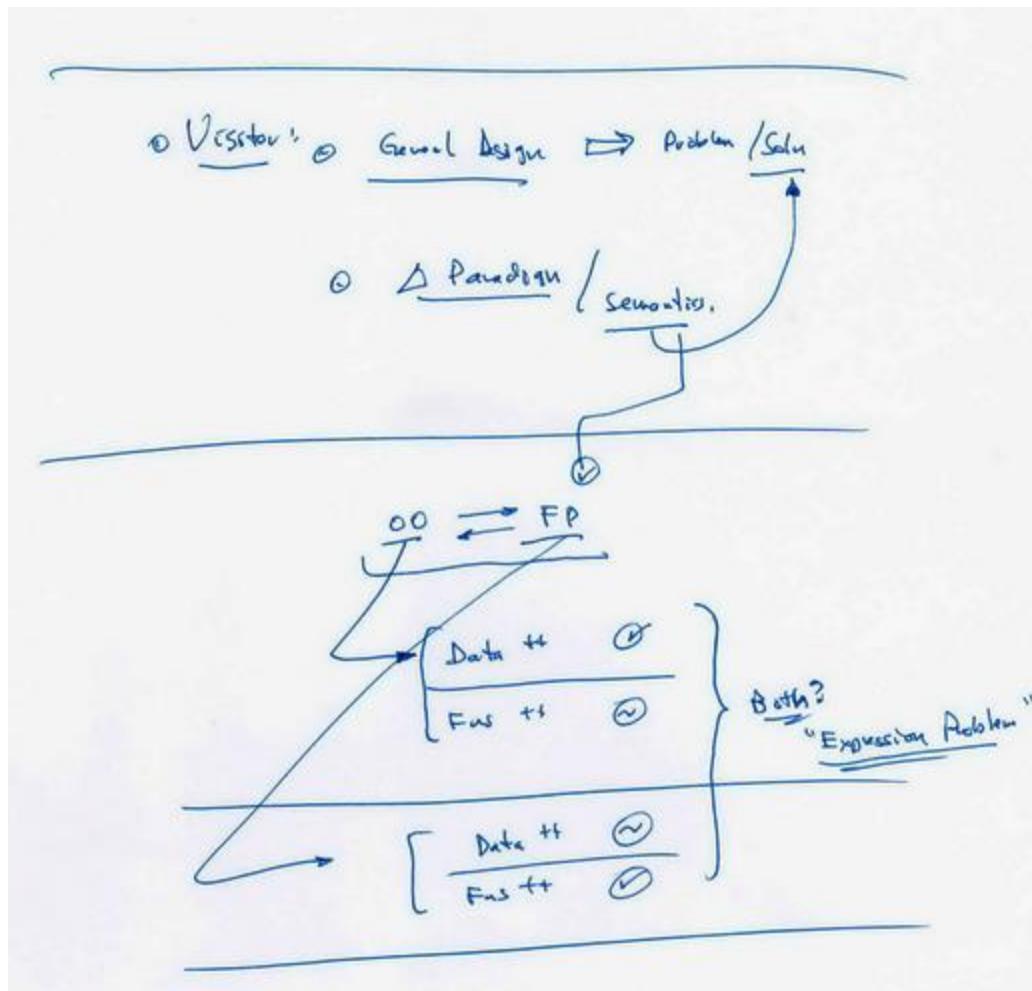


- Some commands are stateful, some are not

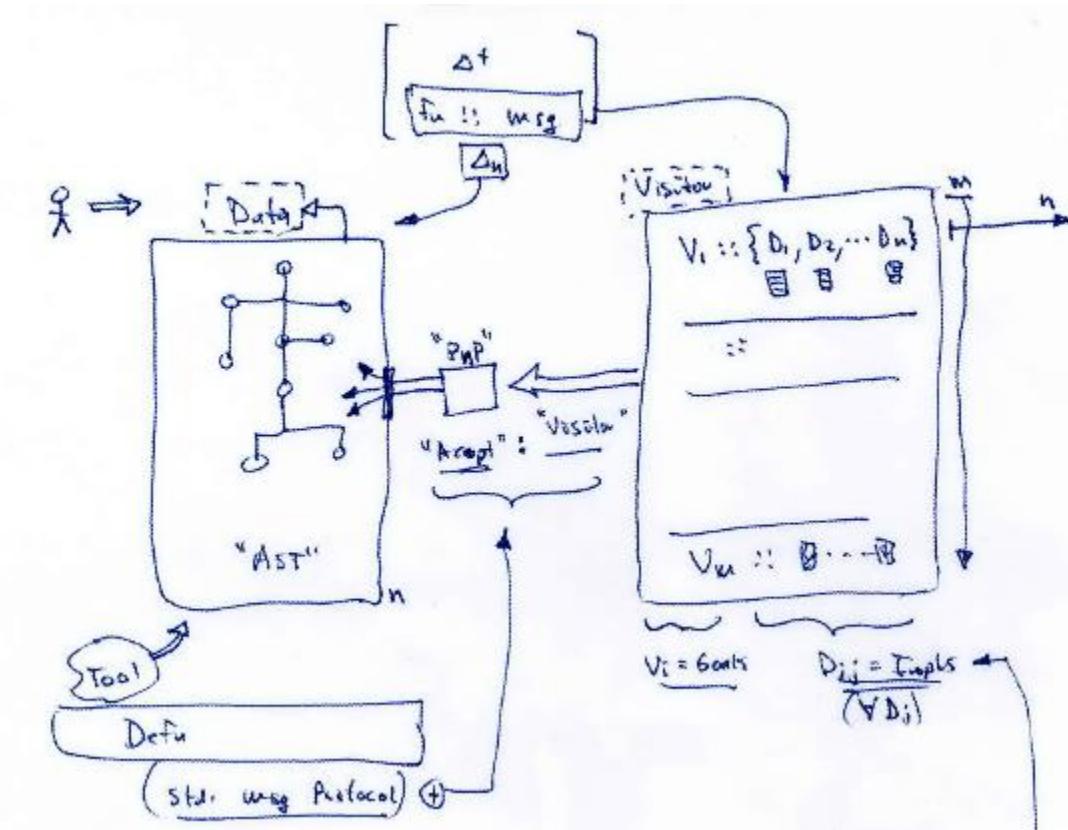


- How to get the results of a command?
Return value? Side-effect? (\approx observer)

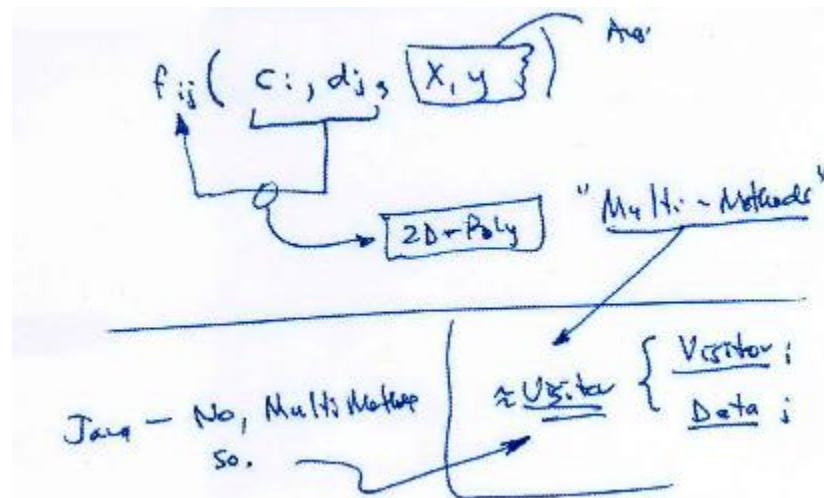
Visitor Pattern::



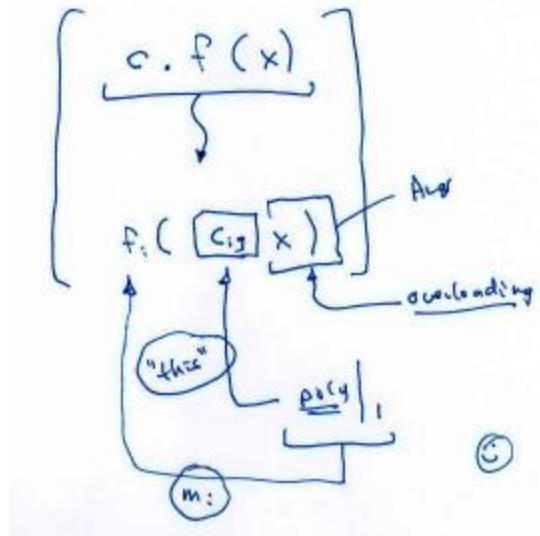
- Visitor Pattern; \approx non-OO \rightarrow Open/closed on functions instead of data



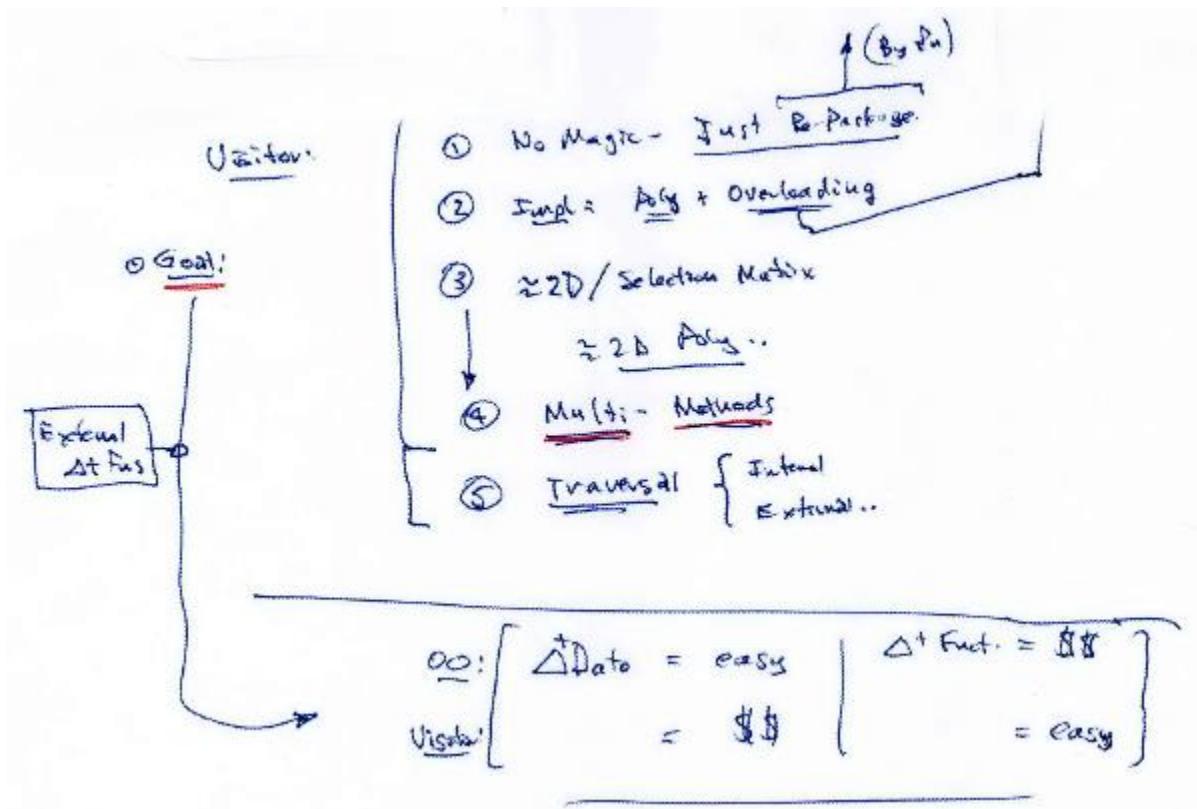
- Architecture of the Visitor Pattern



- One could consider poly dispatch on multiple arguments... \Rightarrow multi-methods



- Standard polymorphism; is single argument poly dispatch



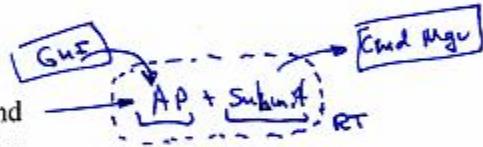
- Summary of results of Visitor Pattern

Day 9: More Command Pattern

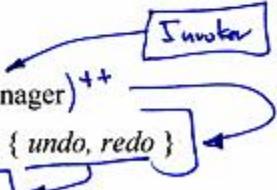
Day 9: More: Command Pattern

- Example code...
 - GUI
 - Switch ...
 - Scheduler
 - ⇒ different *CmdManager* models /methods
 - Event driven (swing)
 - Message driven (switch)
 - Timer driven (scheduler)
 - ...

- *CommandButtons*
 - Combine functions: GUI & Command
(Only works with stateless commands)



- Swing undo/redo
- Lab: Φ_2
 - Review Φ_1
 - Add **Command Manager** Φ_2^{++}
 - Stateful commands { *undo, redo* }
 - *meta-commands* ?
 - GUI mediator



Advanced Software Development

Day 9: More: Command Pattern

- Example code...

- GUI
- Switch ...
- Scheduler
- ⇒ different *CmdManager* models /methods
 - Event driven (swing)
 - Message driven (switch)
 - Timer driven (scheduler)
 - ...

○ CommandButtons

- Combine functions: GUI & Command

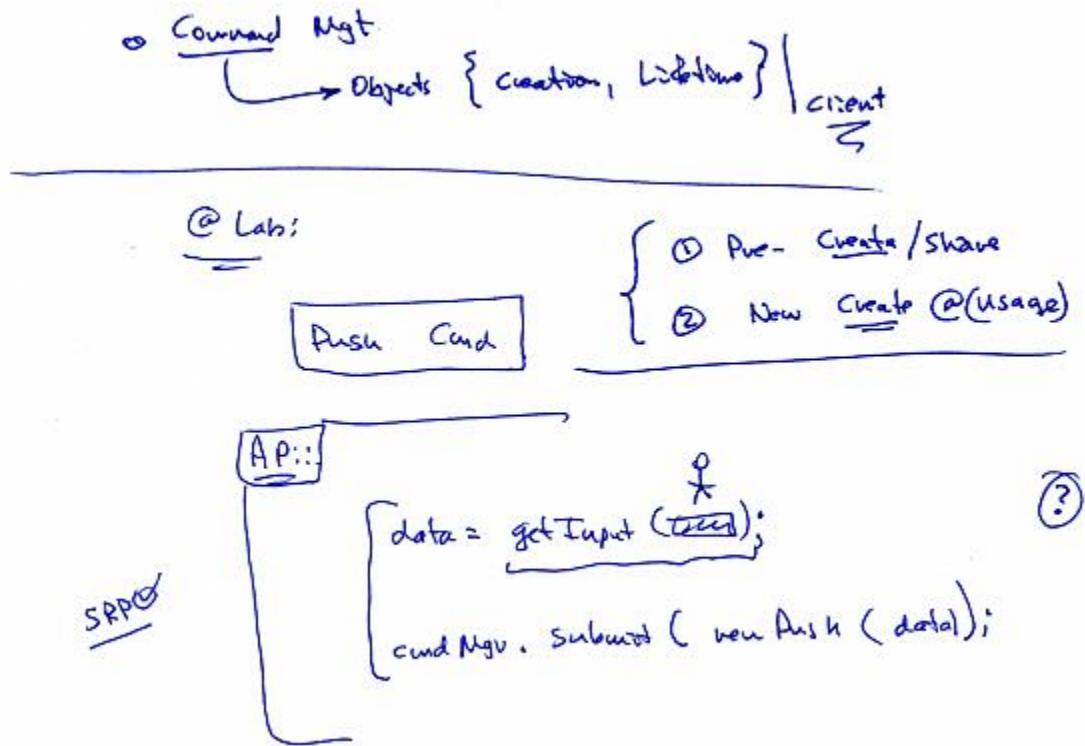
(Only works with stateless commands)

○ Swing undo/redo

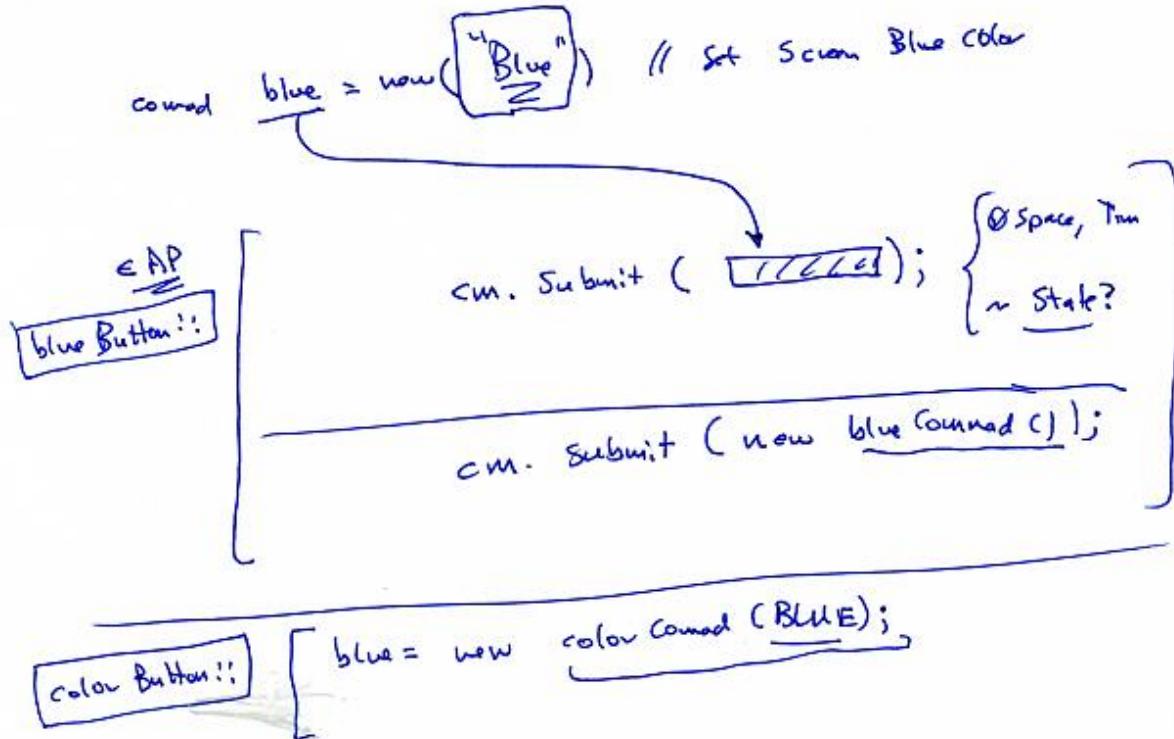
(DIY? → Library)

- Annotated topics(!)

Review of Command Lab – Φ₂



- Design issue of Command object lifetimes and management

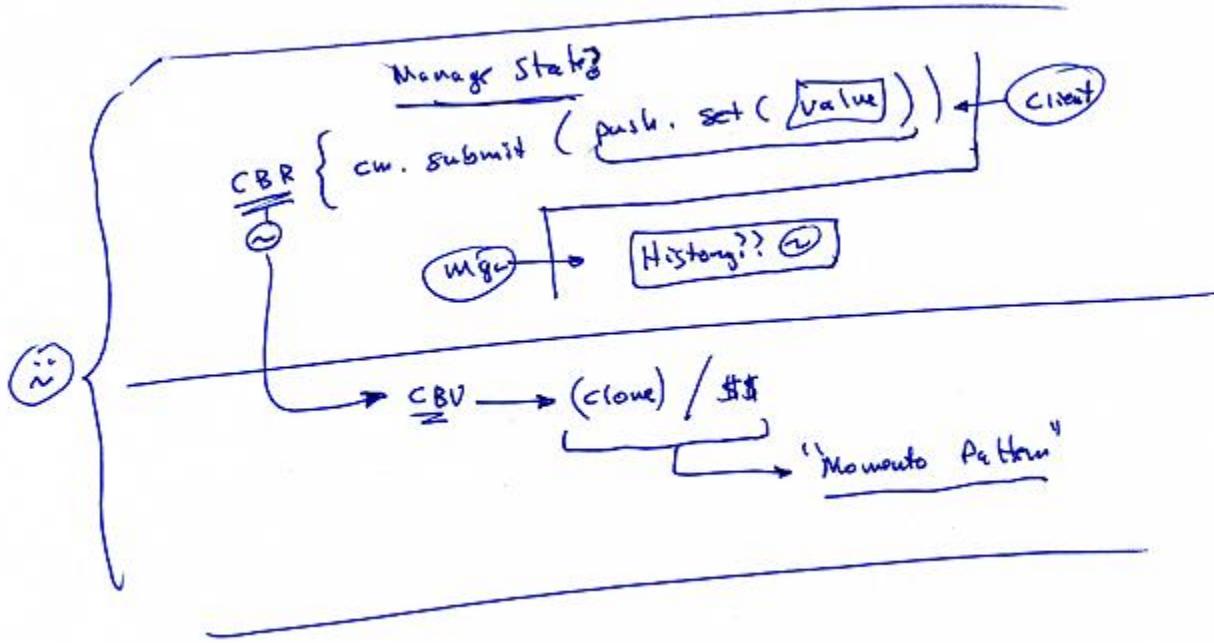


- Example – Simple (stateless) command

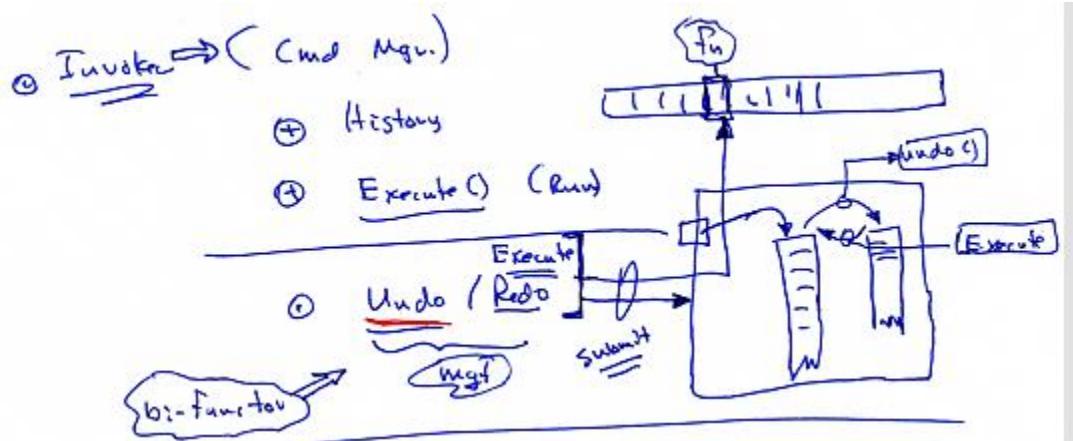
	State?	
	execute	undo
Push	✓	—
Pop	—	✓
Clear	—	vvv
Blue	—	✓

Annotations include "execute", "undo", "State?", "Push", "Pop", "Clear", "Blue", and "@ C:".

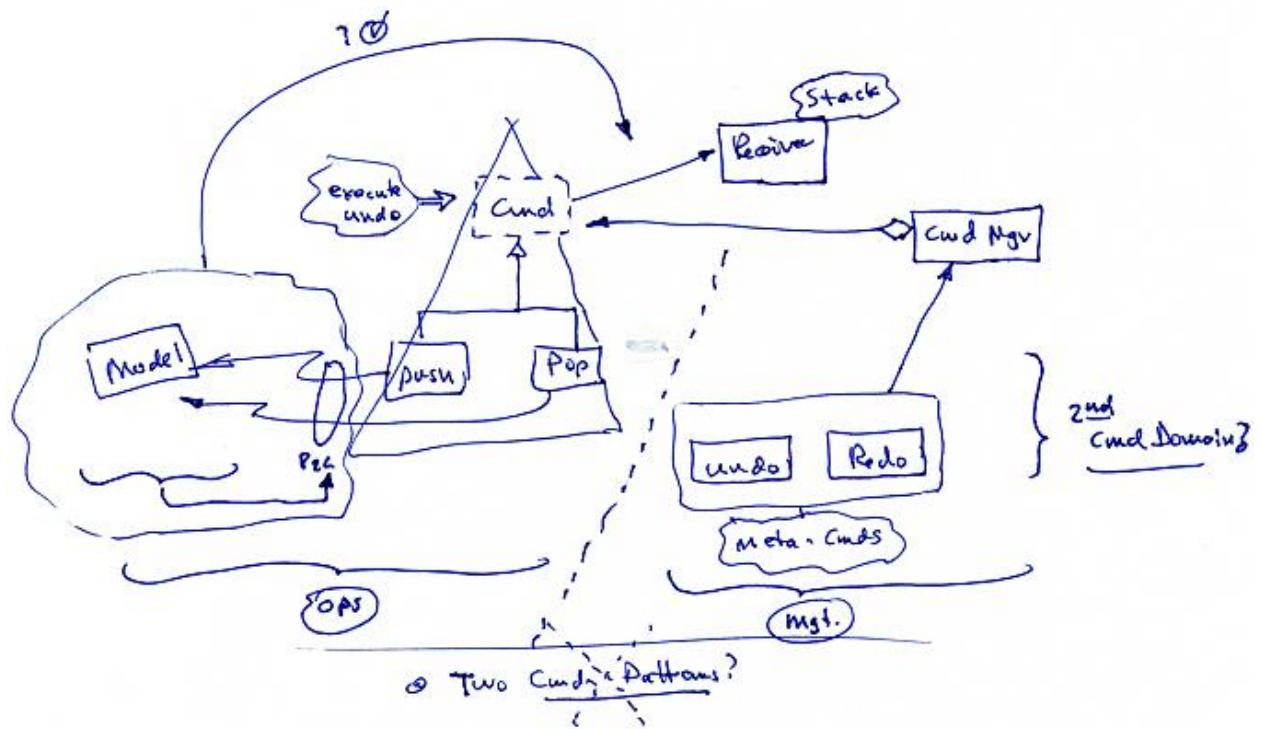
- Stateful commands



- How to manage state in commands?

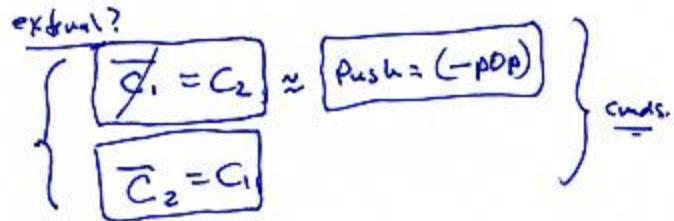


- Converting invoker to Command-manager



- Overall architecture of Lab

⑤ Undo ?



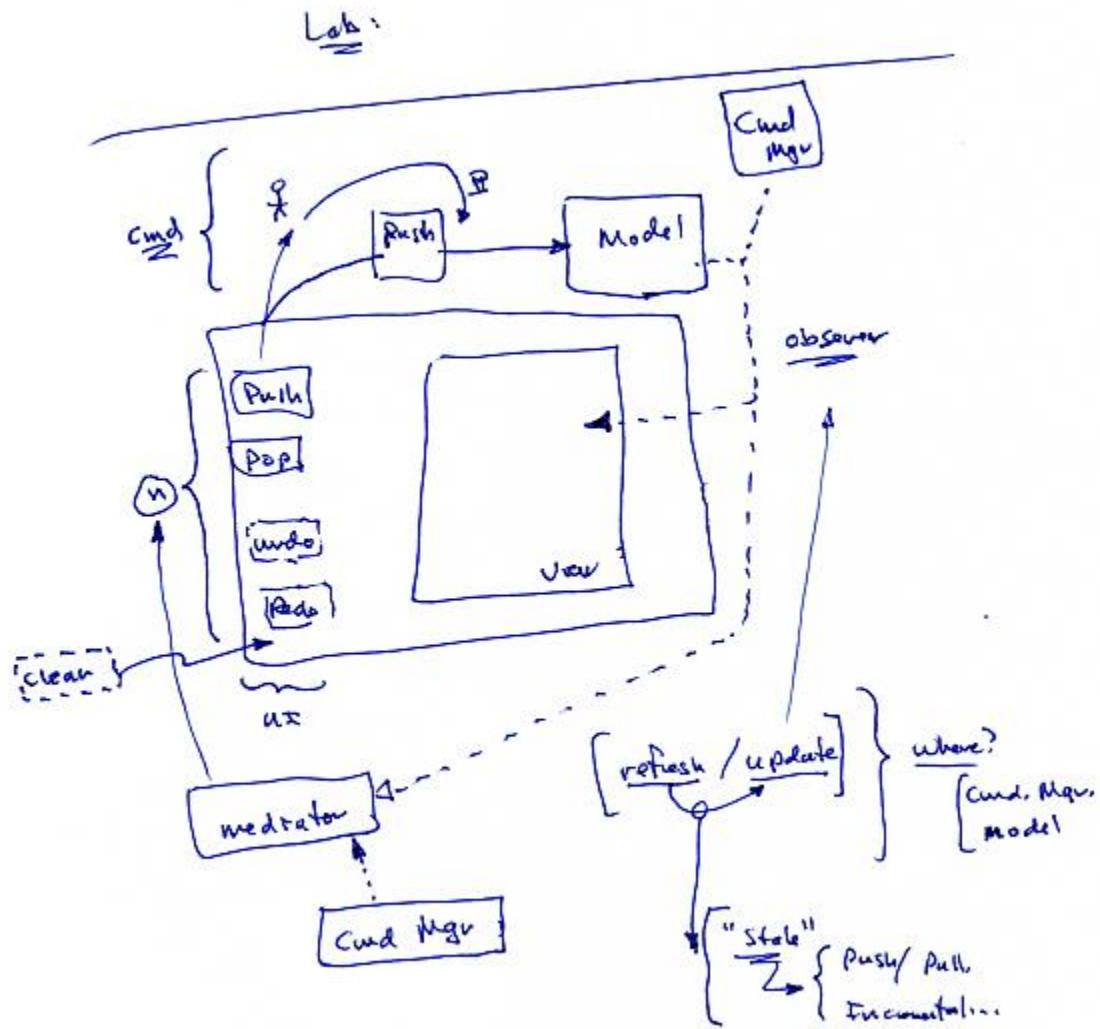
- ⊗ D.N.E. "opposite" cmd
 - ⊗ who would know?
~~(invoker)~~
 - ⊗ modular!
 - ⊗ undo state?
- $\therefore (@\text{cmd})_i$

⑥ undo/redo ; $[\# (\text{cmd} \Rightarrow \text{redo-Cmd})]$

Action Button :: submit (cmd)

msg. " :: msg \rightarrow cmd msg()

- Modularity of undo methods?



- Lab architecture

Swing Command Manager

- Already have sophisticated built-in invoker
Event → GUI dispatch manager → AL/AP
- To add other *cmdManager* capabilities,
We chained the AL/AP's to submit to a secondary CmdMgr
- Swing recognizes the need for this additional capability, so it is (also) built-in.
- Two utility classes are provided;
`javax.swing.undo.UndoManager & RedoManager`
- Example;

To add undo & redo buttons to a JTextArea:

```
UndoManager manager = new UndoManager();
textArea.getDocument().addUndoableEditListener(manager);

JToolBar toolbar = new JToolBar();
toolbar.add(UndoManagerHelper.getUndoAction(manager));
toolbar.add(UndoManagerHelper.getRedoAction(manager));
```

Well, plus about another 1½ pages of supporting setup. ☺

- Swing supports undo/redo for its standard components

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.undo.*;
import javax.swing.text.*;
import javax.swing.event.*;

public class UndoAndRedoOperations extends JFrame {
    JButton b1, b2, b3;
    JTextArea area;
    JScrollPane pane;
    JPanel p;
    UndoManager manager = new UndoManager();
    public UndoAndRedoOperations() {
        p = new JPanel();
        area = new JTextArea(5, 30);
        pane = new JScrollPane(area);
        manager = new UndoManager();
        b_undo = new JButton("Undo");
        b_redo = new JButton("Redo");
        b_exit = new JButton("Exit");
        b_undo.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { manager.undo(); }
                catch (Exception ex) { }
            }
        });
        b_redo.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { manager.redo(); }
                catch (Exception ex) { }
            }
        });
        b_exit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        area.getDocument().addUndoableEditListener(
            new UndoableEditListener() {
                public void undoableEditHappened(UndoableEditEvent e) {
                    manager.addEdit(e.getEdit());
                }
            }
        );
        p.add(pane);
        p.add(b1);
        p.add(b2);
        p.add(b3);
        add(p);
        setVisible(true);
        pack();
    }

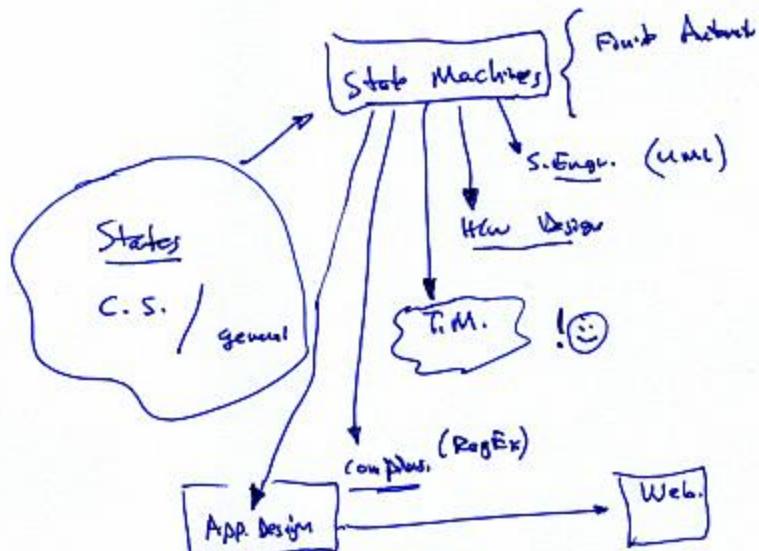
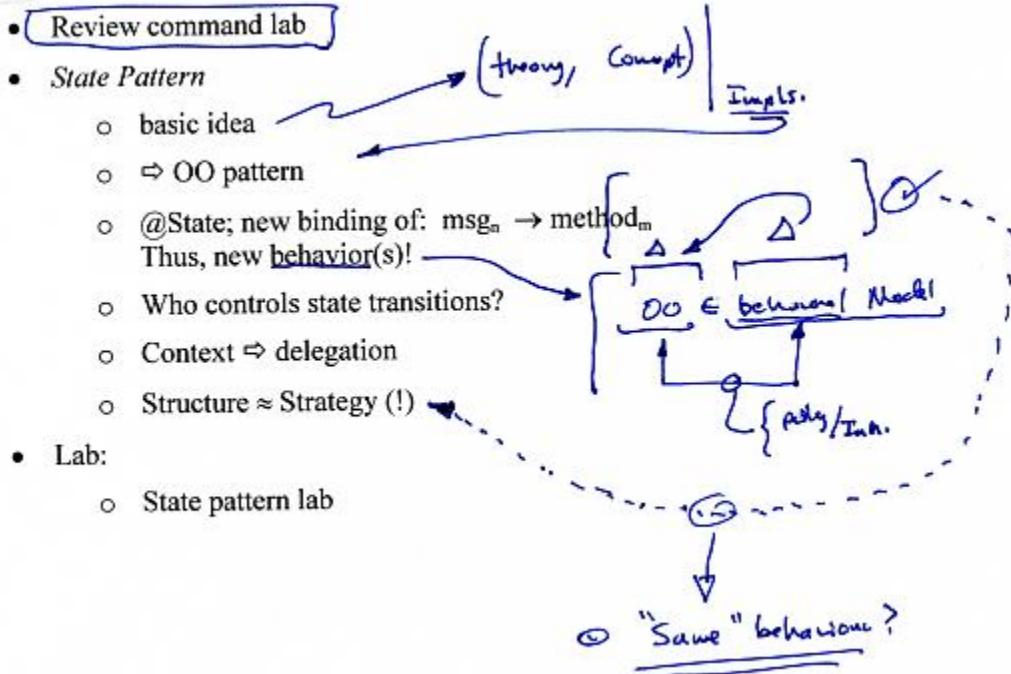
    public static void main(String[] args) {
        UndoAndRedoOperations op = new UndoAndRedoOperations();
    }
}

```

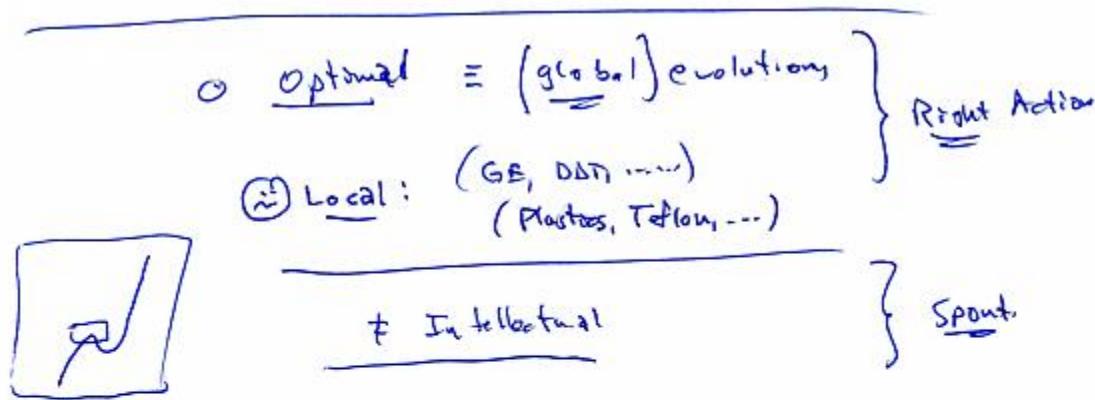
D.I.Y. Semantics → Known Semantics | Model

- Swing Example

Day 10: State Pattern



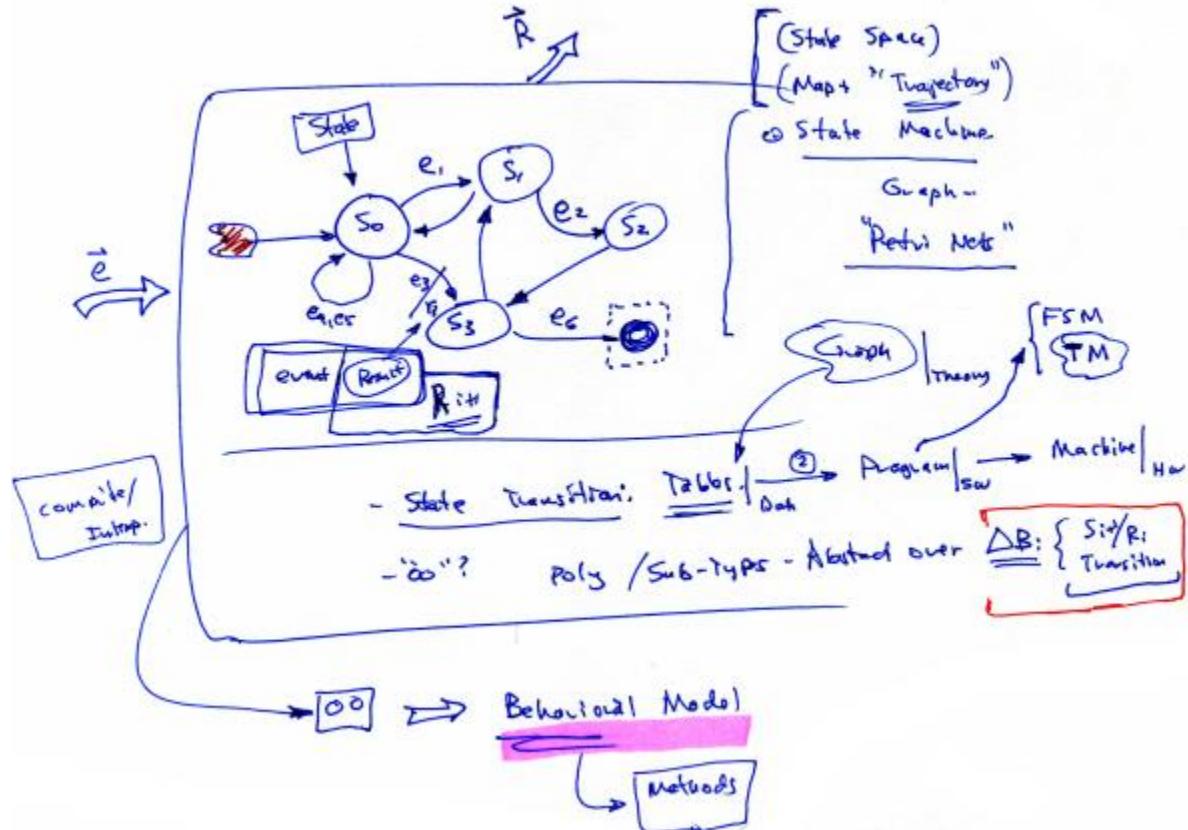
- State machines are a general idea in Computer Science



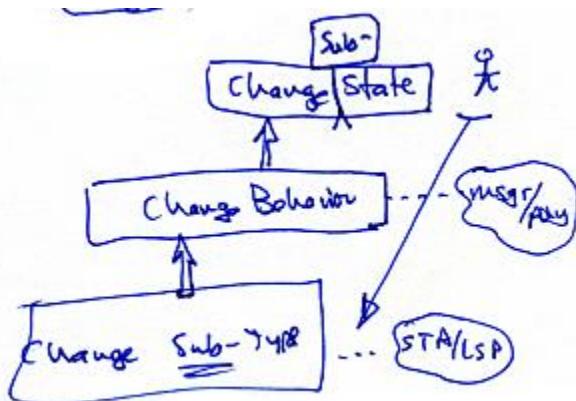
$$\underline{\text{Best Good}} = \text{Global} / \text{Local}$$

$\circ \underline{\text{Behavior}} \leftrightarrow \underline{\text{State of CSC}}$

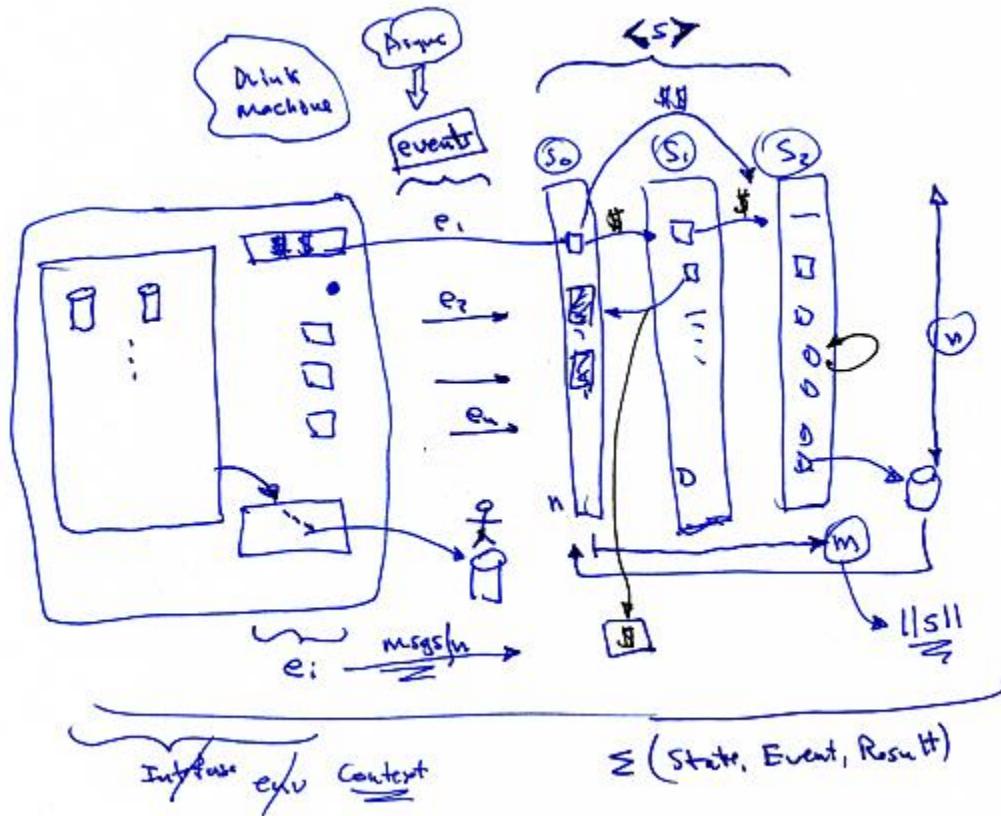
- Optimal behavior (Right Action) is not intellectual, only possible from higher states of consciousness



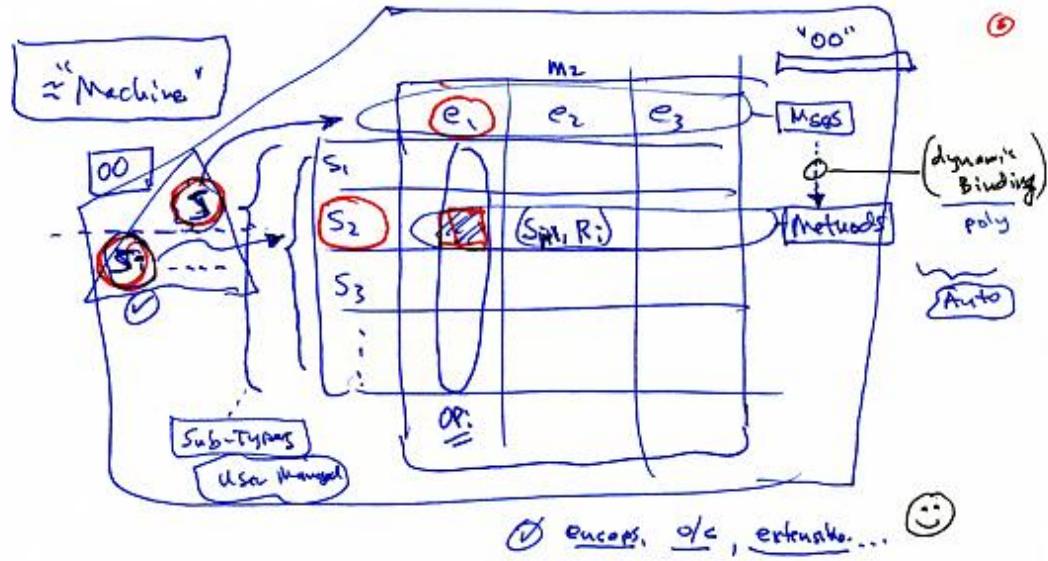
- State space graphs are a general model



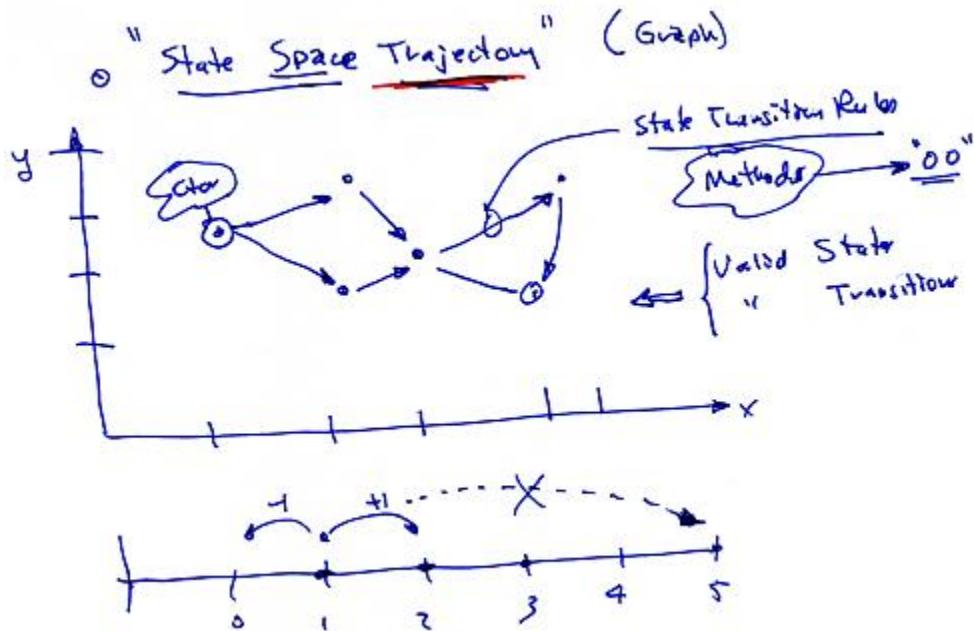
- The OO model implementation



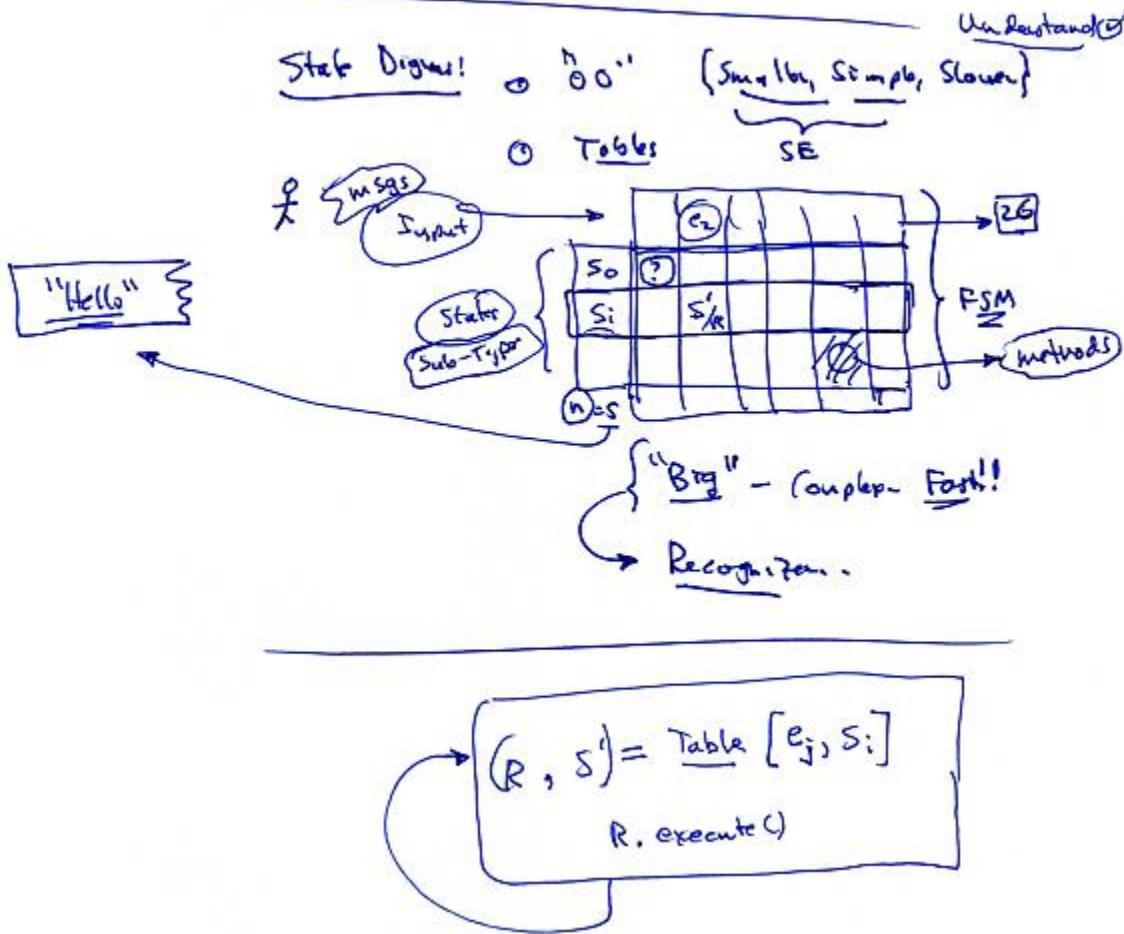
- Example: A Vedic (organic) juice machine (!)



- One can specify the state model in a table, and then map that to an OO model



- The trajectory describes the valid state transitions
- Other points or paths may be possible, but not valid (legal)

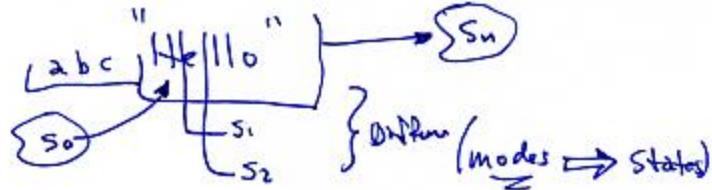


- For more complex (big) applications, a table driven state machine would be faster, but harder to understand and create
- Thus there are tools to do this automatically for those applications (compilers, lexers, Regex, ...)

State Machine Examples :

② Savon Gant - StageIn

Regex



map → Graph → Table → "00"

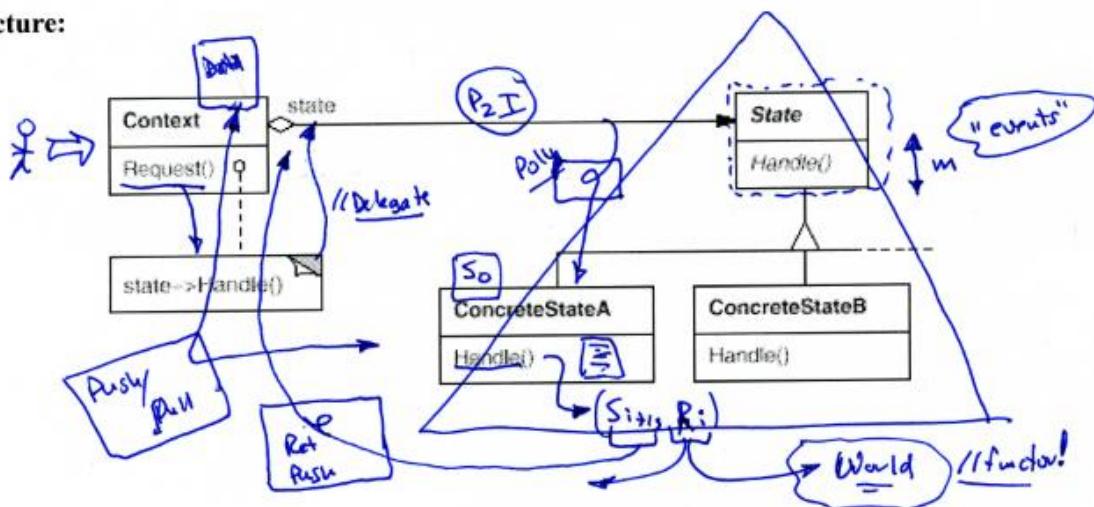
② Lock.

③ Vending Machine,

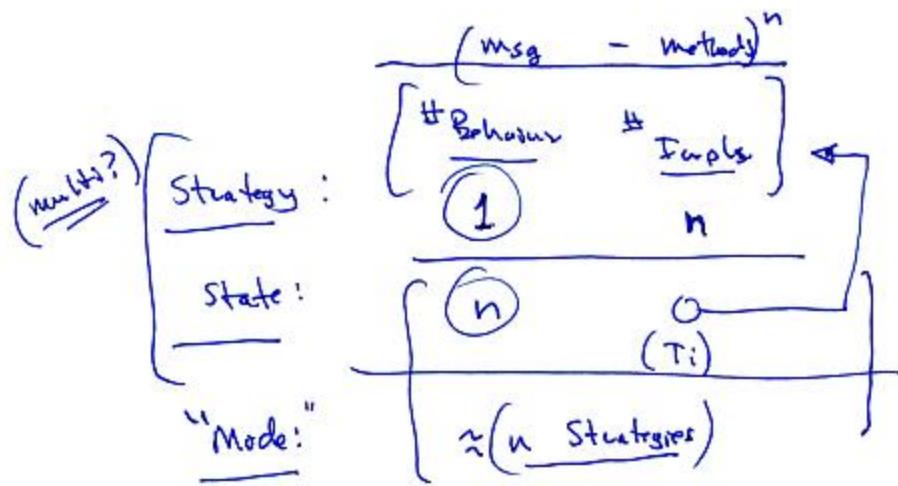
- Examples of state machines

State Pattern

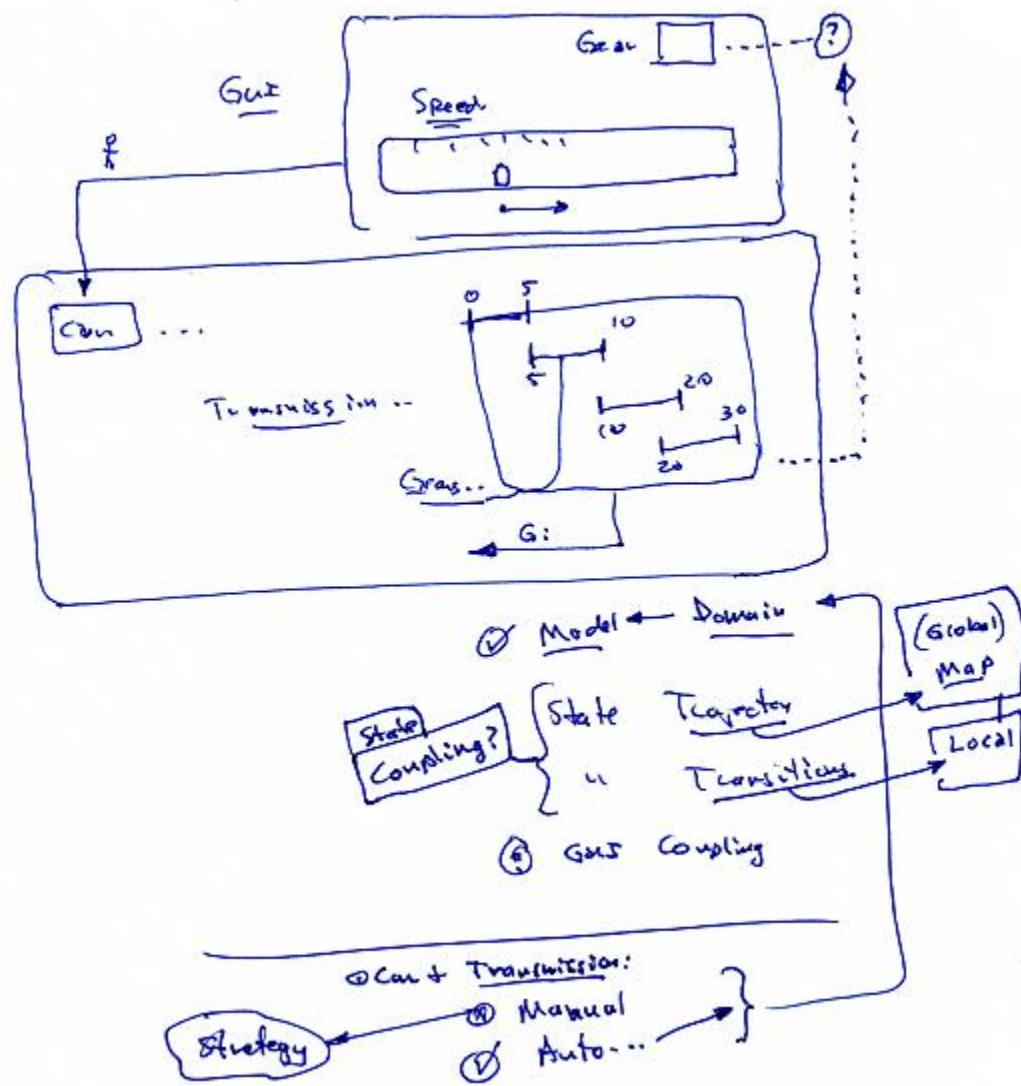
Structure:



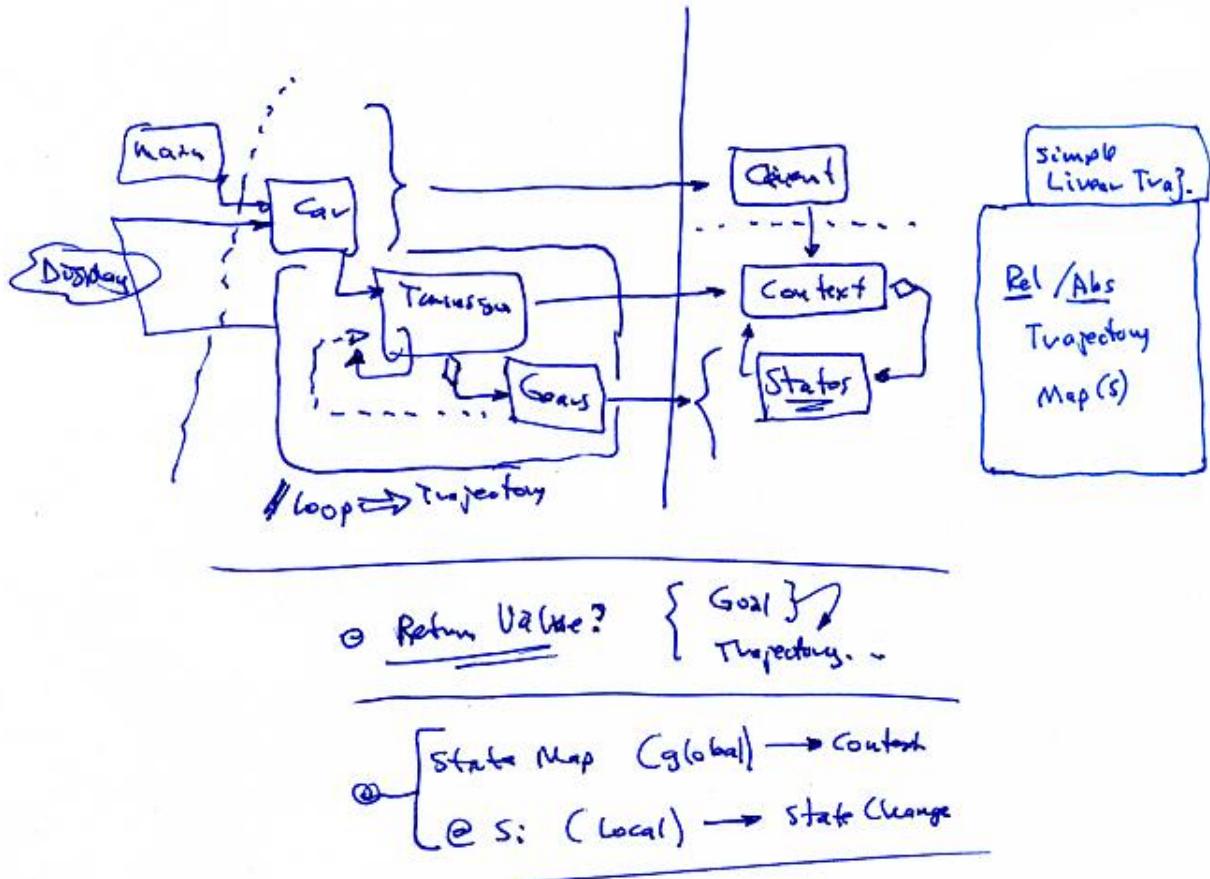
- GOF model example



- Compare to strategy pattern
- → State pattern uses strategy pattern...



- State pattern lab



- State pattern lab review

Day 11: Proxy & COR Pattern

Daily Topics

Day 10:

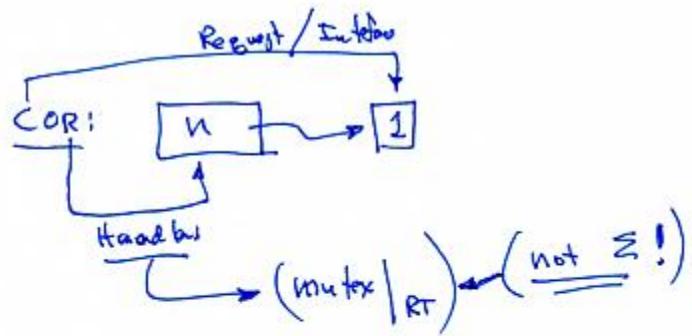
Proxy, COR Patterns

- Quiz & Review
- Review state lab
- Proxy Pattern
 - important concept
 - uses, impacts
 - ⇒ OO pattern
 - See: GOF
- Proxy improvements, implementations
 - Functor, generics
 - Dynamic proxy (later...)
- Examples
- Chain of Responsibility pattern
- Review Session for Midterm Exam
- Next;
 - *dynamic proxy*

*(proxy ≈ Decorator)*SE → { Abstraction
Re-use }{ + Flexible
- Perf, Correctness }

Reflection

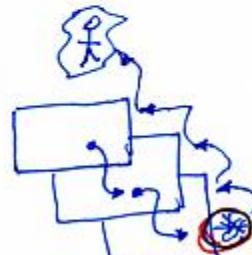




④ Exceptions:

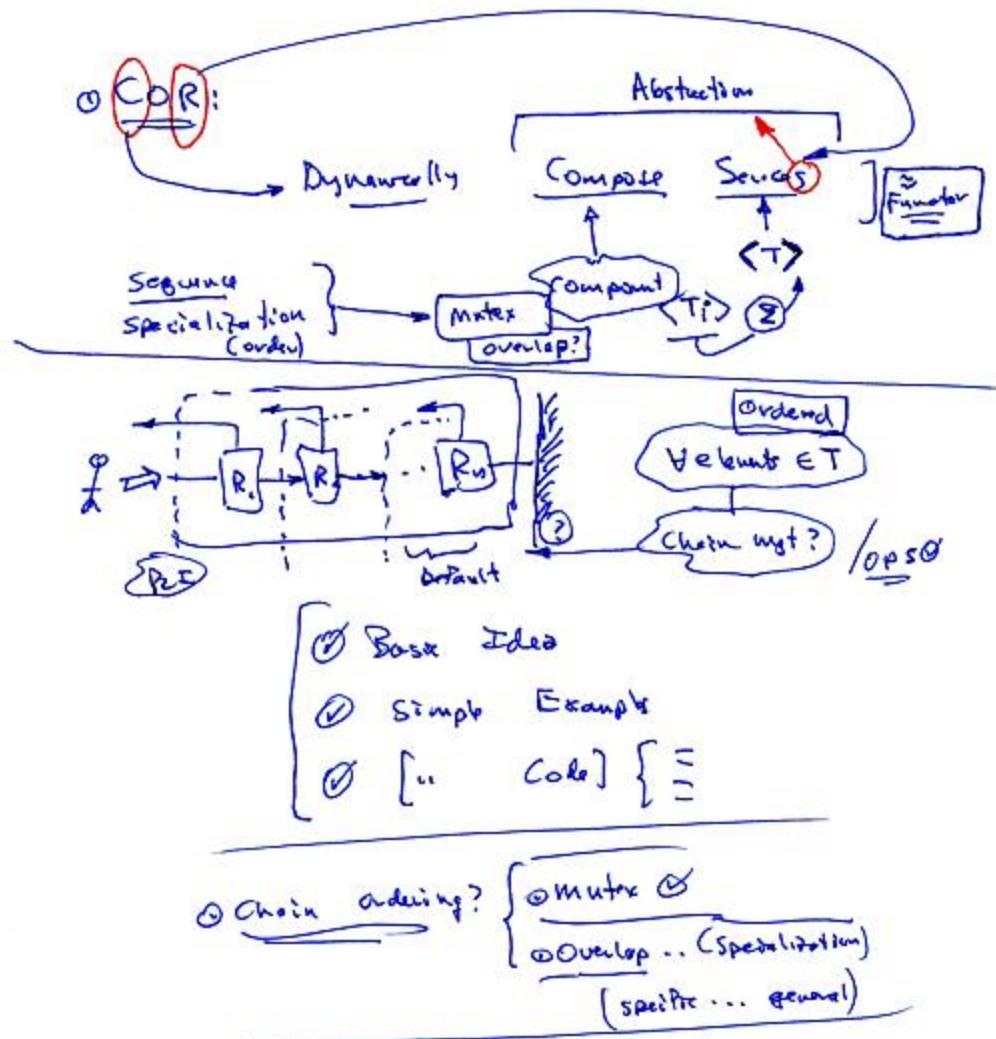
Hierarchy of Handlers

$\left[\begin{array}{l} \text{(≈ chain)} \\ \text{(seq./order)} \end{array} \right]$

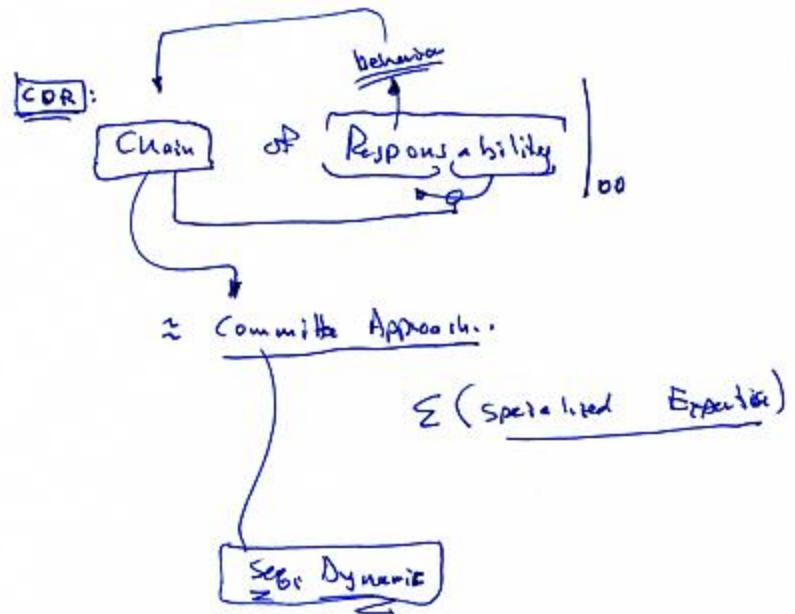


④ Two levels of Nesting (CT, RT)

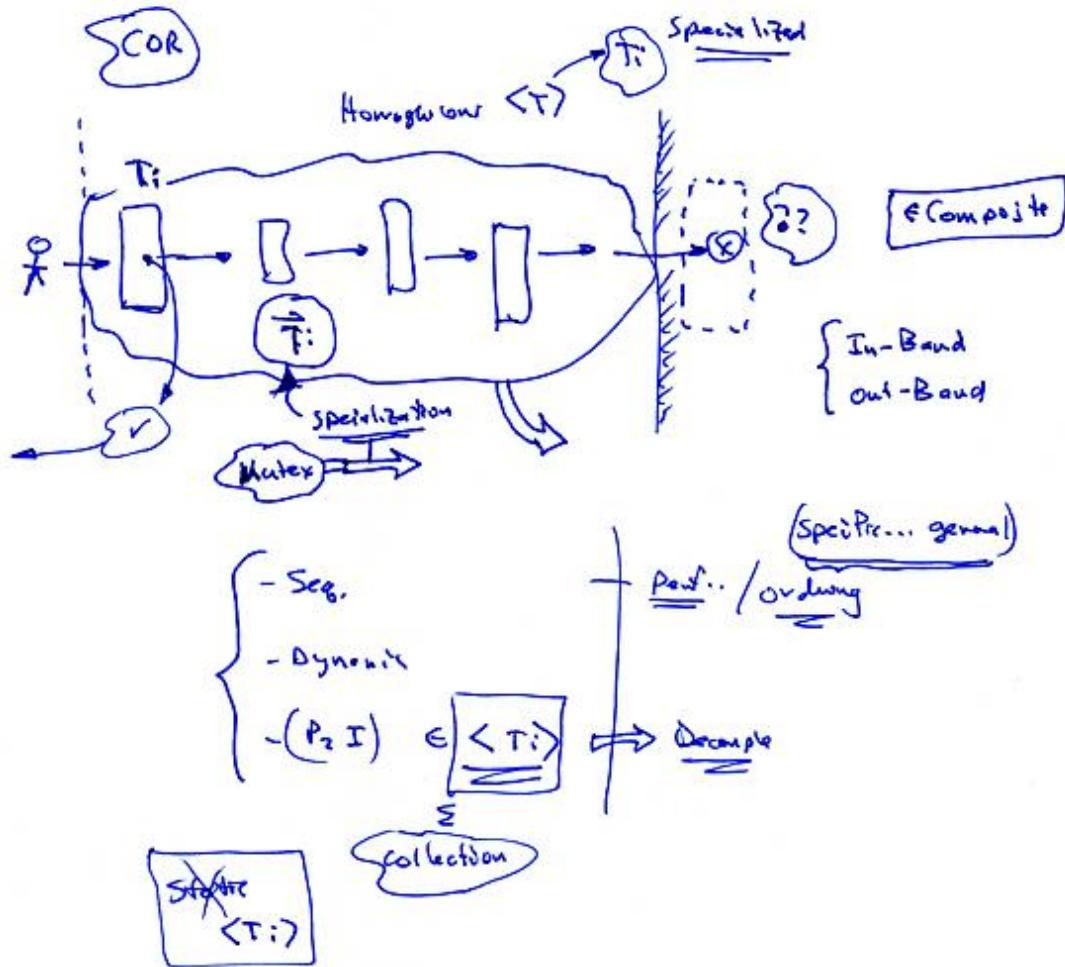
- COR pattern is already familiar, Exception hierarchies
- They are linear, hierarchical (ordered)
- Also they have both static (CT, definitions scopes) and dynamic (RT, function call hierarchies) structure



- Basic idea of COR pattern



- COR ideas

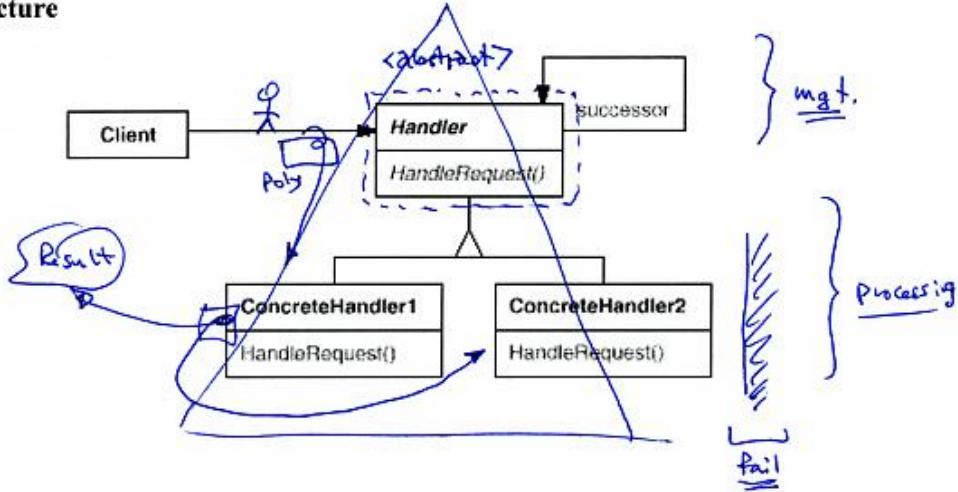


- COR Structure

Chain of Responsibility

Intent $(R^* = \sum R_i)$ R: Decouple the sender of a request and its receiver by creating a pool of potential receivers, and giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

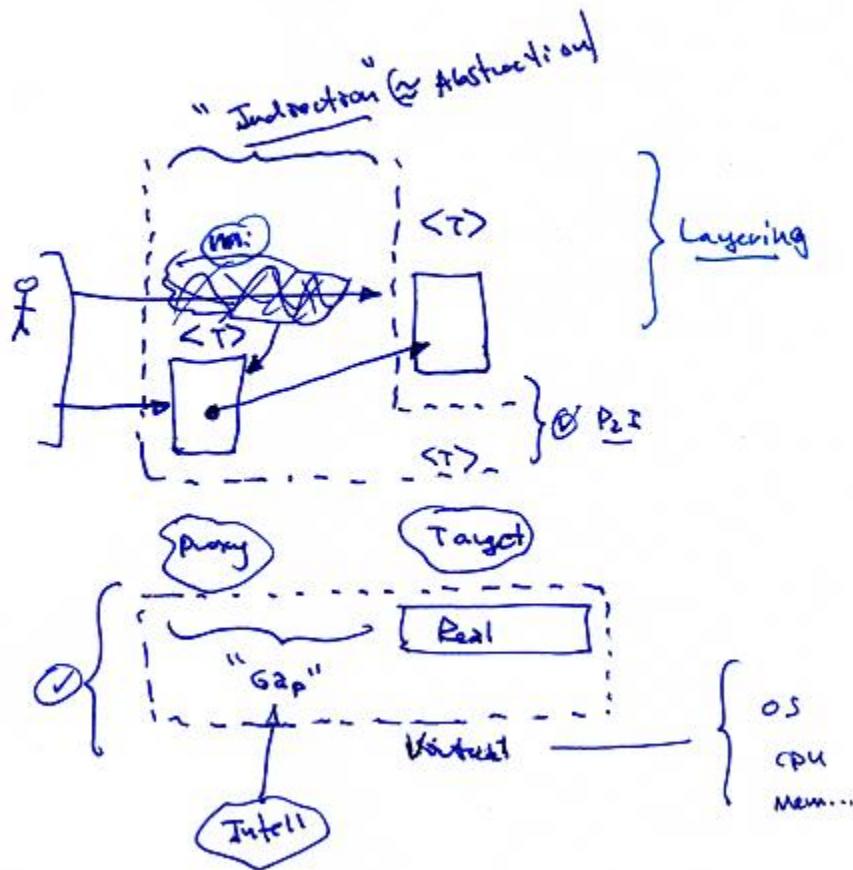
Structure



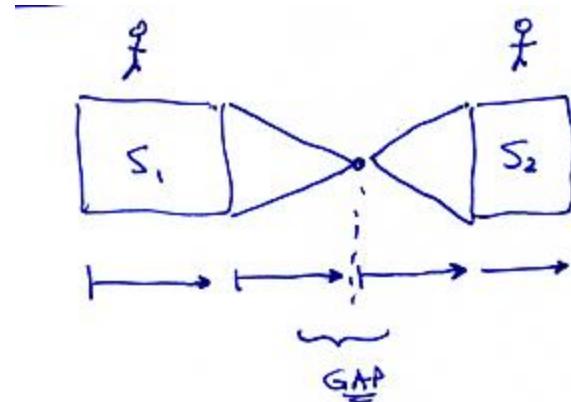
- GOF description

Proxy Pattern::

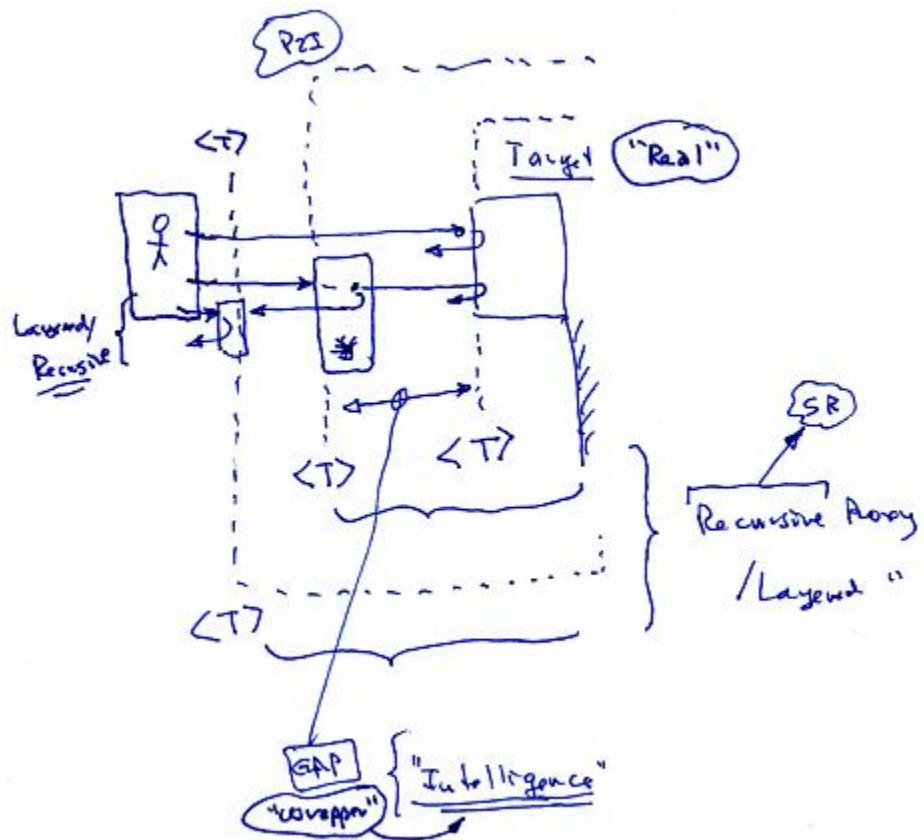
① Proxy ≈ "Surrogate"



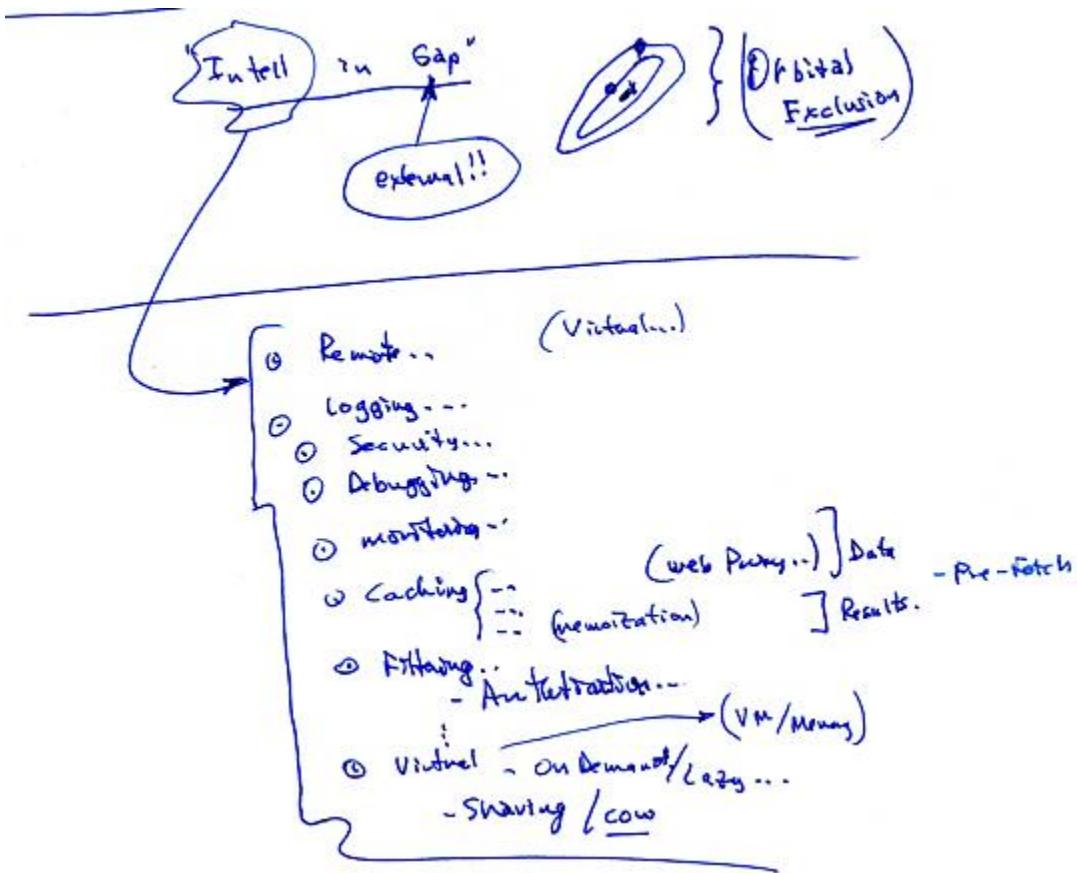
- General idea of Proxy Pattern



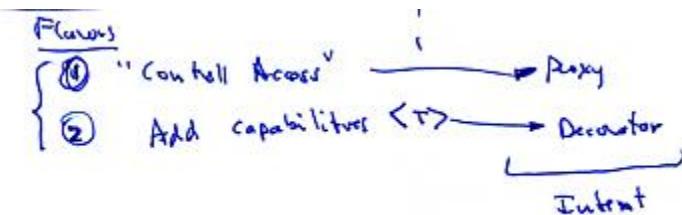
- SCI describes the model of the transcendental “intelligence in the gap”



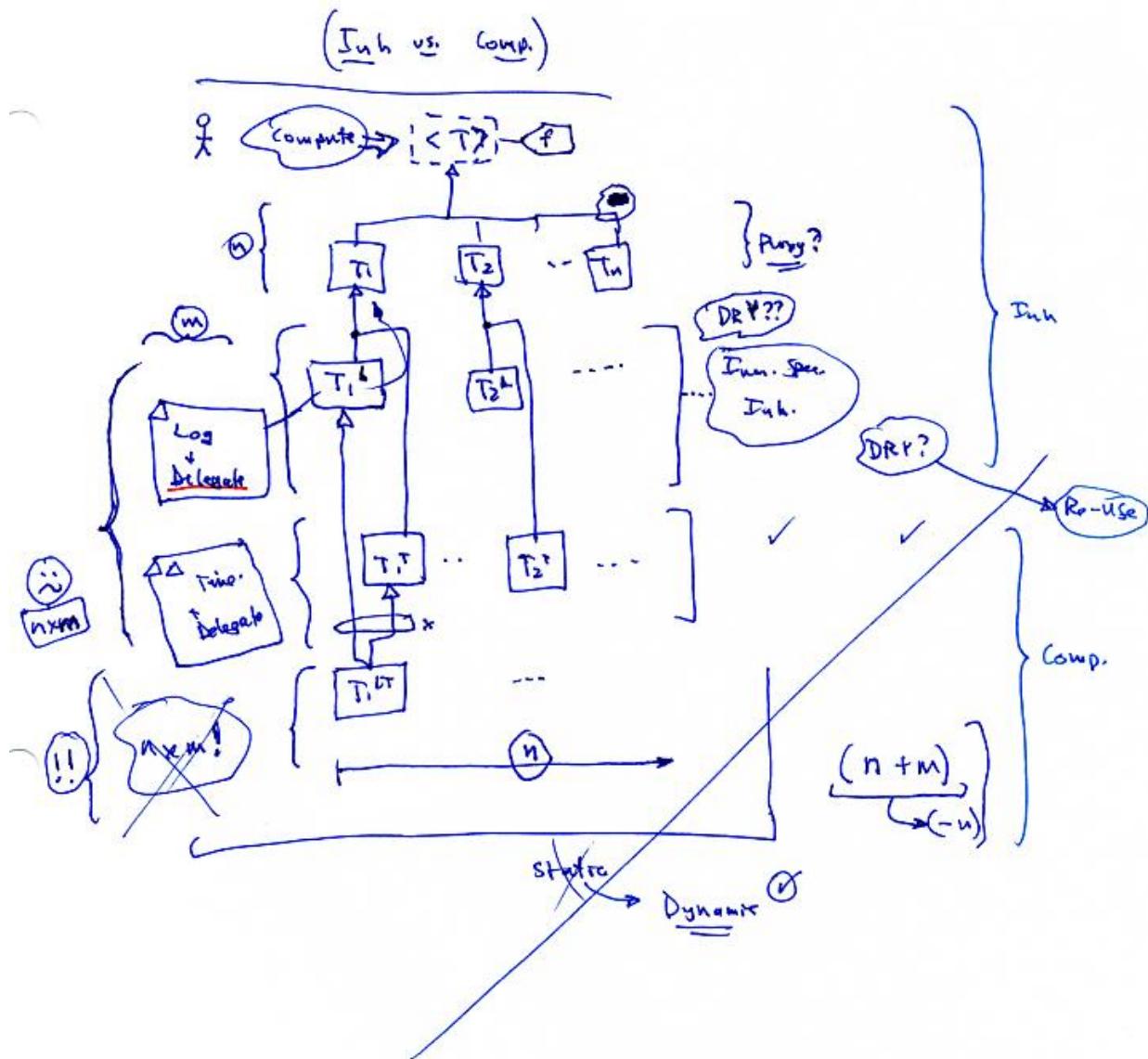
- Model of the proxy pattern



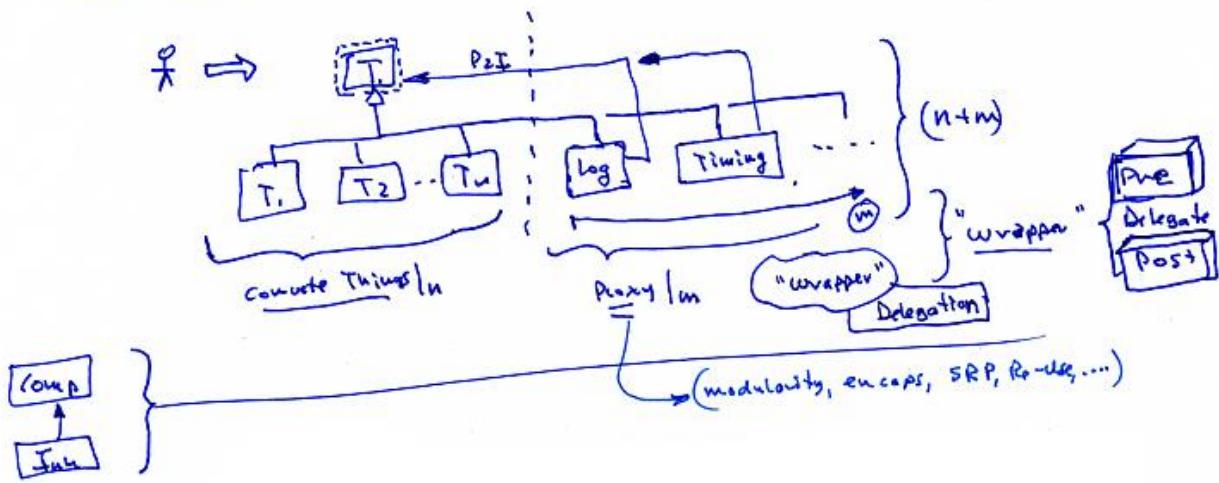
- Applications of Proxy



- Decorator is related to proxy; same structure, different intent



- Implementation choices for proxy wrappers; composition and inheritance
- Inheritance model is not so good...

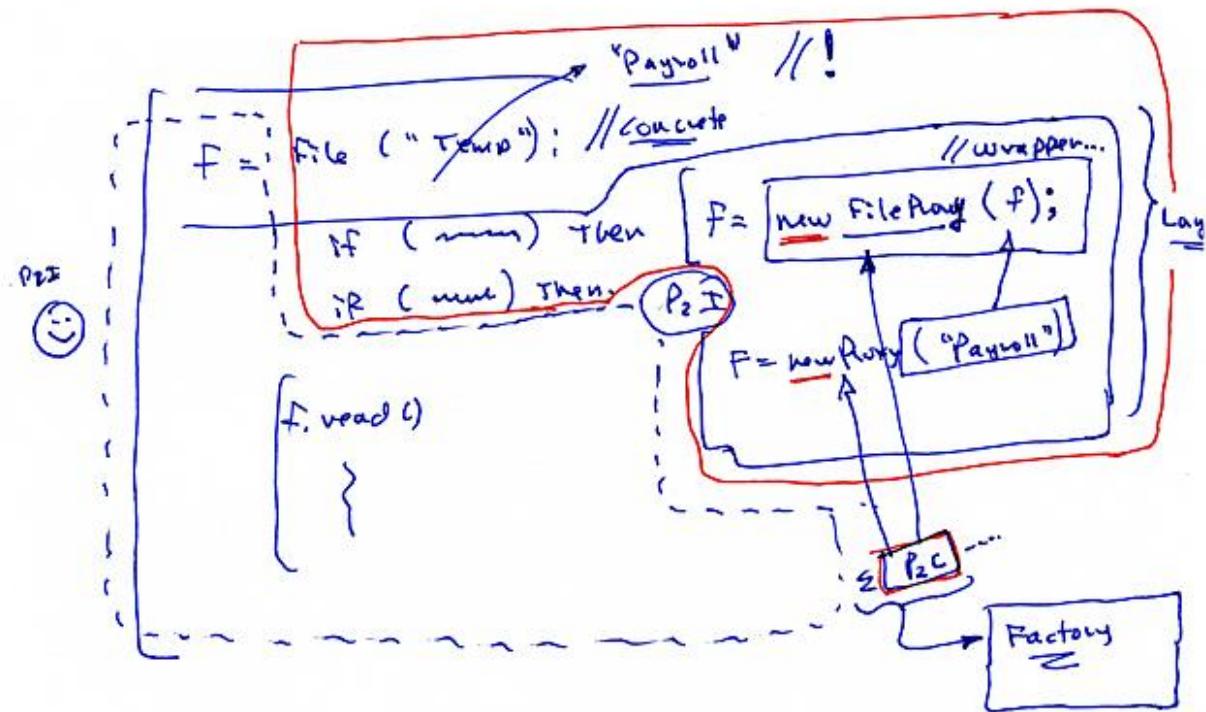


- Composition model \Rightarrow Good!

• Proxy: Simple (SRP, DRY)
 composition $(m+n)$

Dynamic
 Transparent

- Result of using composition model for proxy



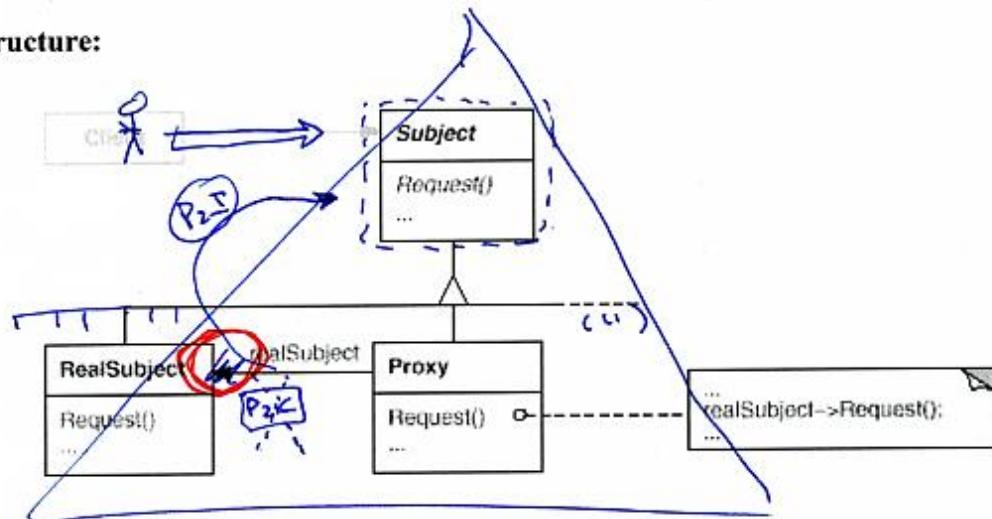
- Proxy allows transparent (P₂I) substitution of wrapped (proxy) objects for “real”

Proxy Pattern

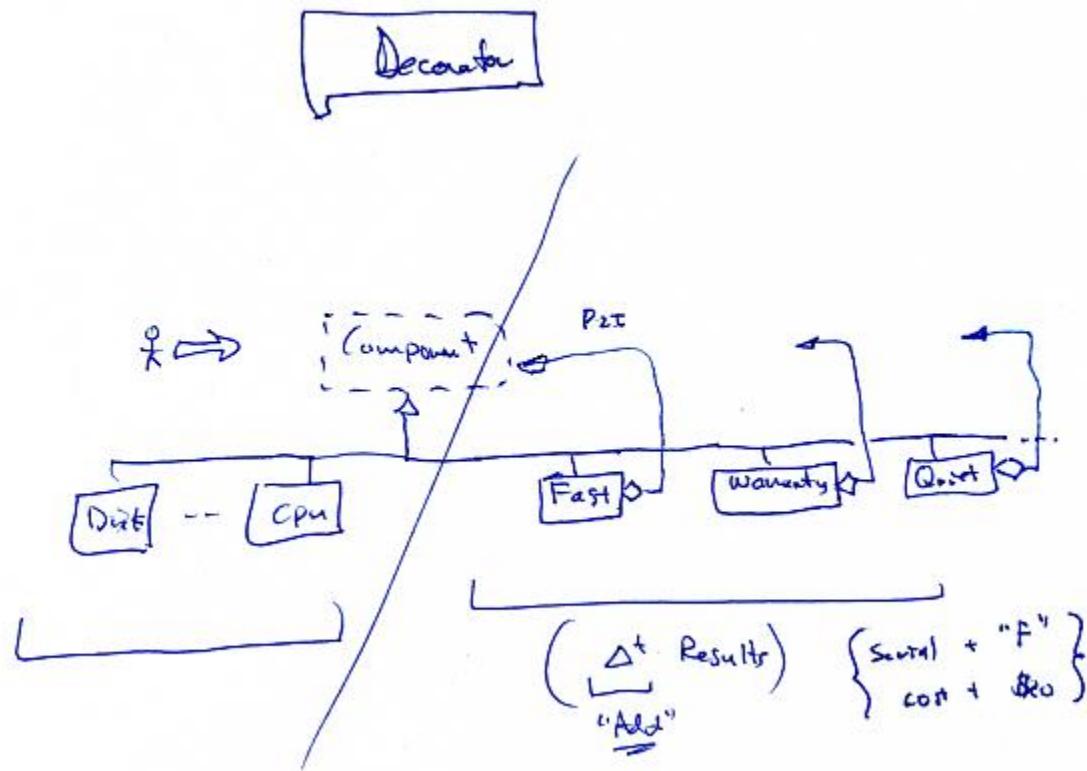
Intent:

Provide a surrogate or placeholder for another object to control access to it.

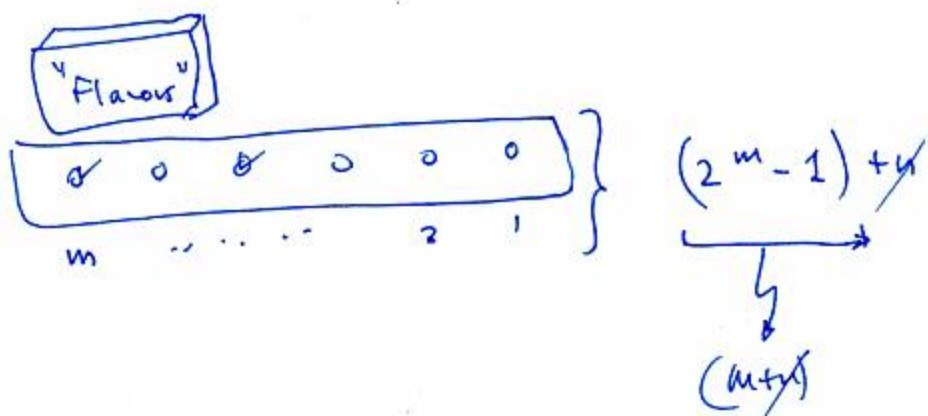
Structure:



- GOF description



- Decorator pattern has same structure
- Difference is in the use, to “*add capabilities*”
- This means changing the results of the delegated messages



- Counting the number of static classes needed for inheritance model for proxy (or decorator)

cs525: Pattern Notes
 Transparent?!

Intent:

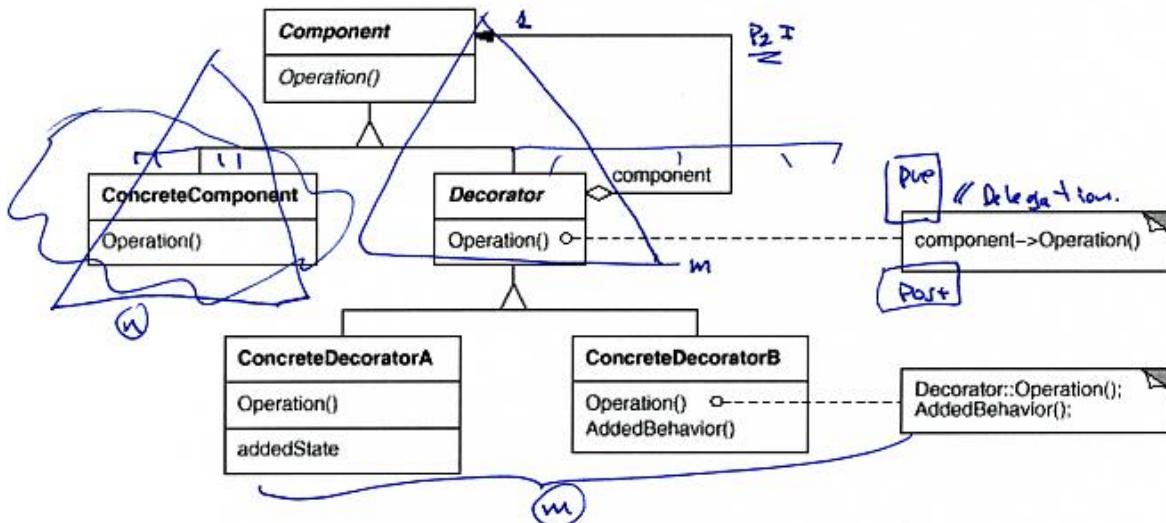
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Dynamis

(ntm)

Decorator Pattern

Structure:

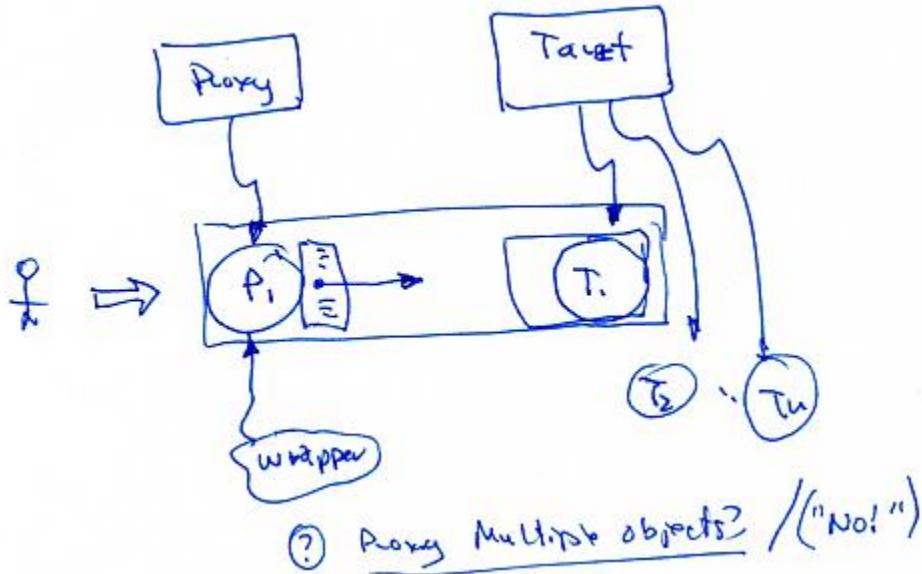


- GOF Description

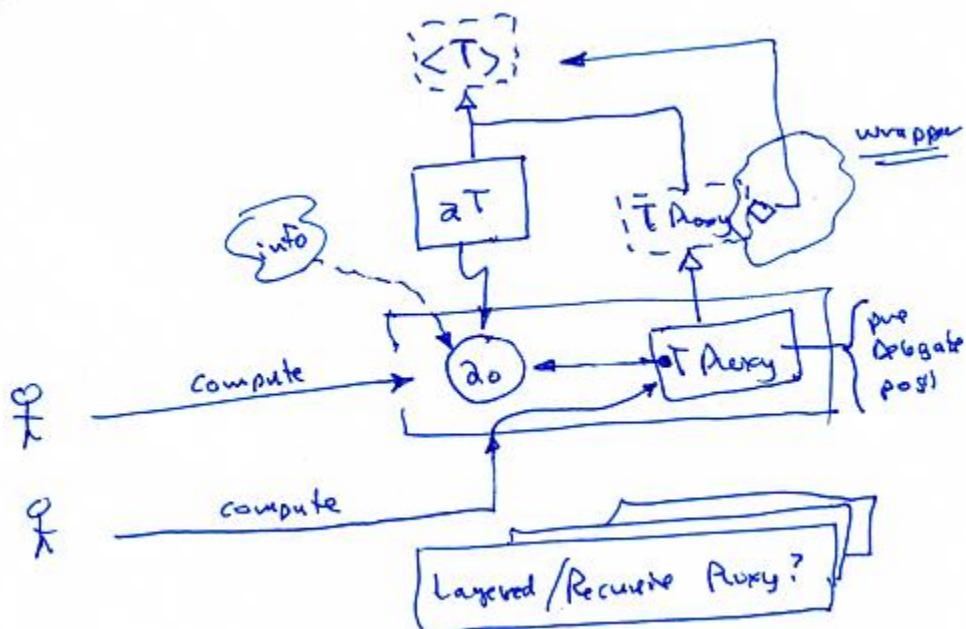
Day 12: More Proxy; Dynamic & Generic Proxy

Day 12: Proxy Fanciness, & Dynamic Proxy

- Review *Proxy Pattern*
 - dynamic layers of intelligence
 - ~~n^2 (m^2)~~ static → (n+m) dynamic
 - Examples:
 - Memoization proxy
 - **Proxy⁺⁺**
 - increased generality through abstractions
 - derived proxy classes
 - override wrapper methods
 - use template pattern, create *pre()* & *post()* methods
 - ⇒ method abstraction
 - functor proxy
 - ⇒ functional abstraction
 - generic functor proxy
 - layered Proxies
 - proxy factory
 - Results:
 - dynamic, reuse, abstraction
 - Exercise:
 - write simple logging proxy
 - *dynamic proxy*
 - motivation
 - implementation
 - examples
-



- Review question:
Can one proxy multiple objects? (No.)



- From Proxy exercise; recall that proxy is an envelope, and the user provides the contents!

① Proxy Exercise: $(\text{Generalize}, \text{Extend})$

① Simple Proxy

Log Proxy (= Thing Log Proxy)

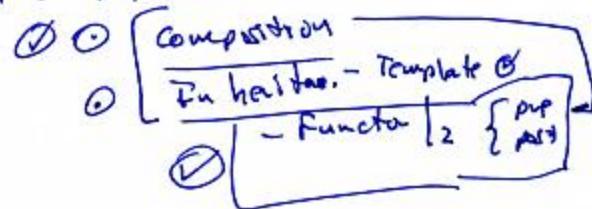
② Extend - Multiple Thing - Type

Thing <T> Generalize

Log Proxy <T> extends Thing <T>

③ Allow Different Actions -

Two Approaches:



③ Generalize over Target Types

④ No. ⑤ Structure of Proxy

Depends on " " " Target

[Count + Signatures of Methods]

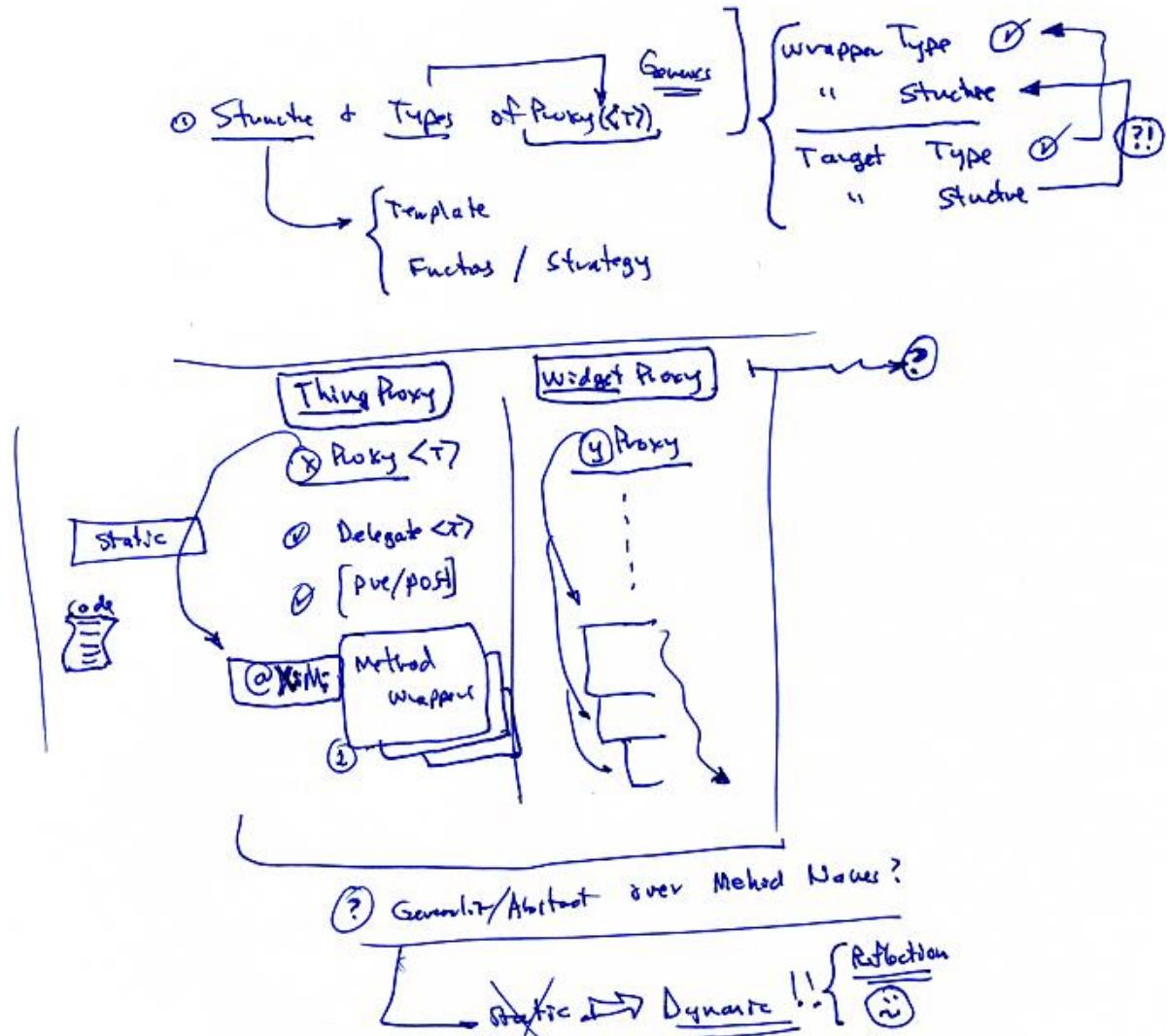
Well, ...

① Text Edit / Transform

② Dynamic Proxy (G Reflection!)



- Proxy exercise, and follow-up lab



- Summary of issues with trying to generalize a proxy over target object types

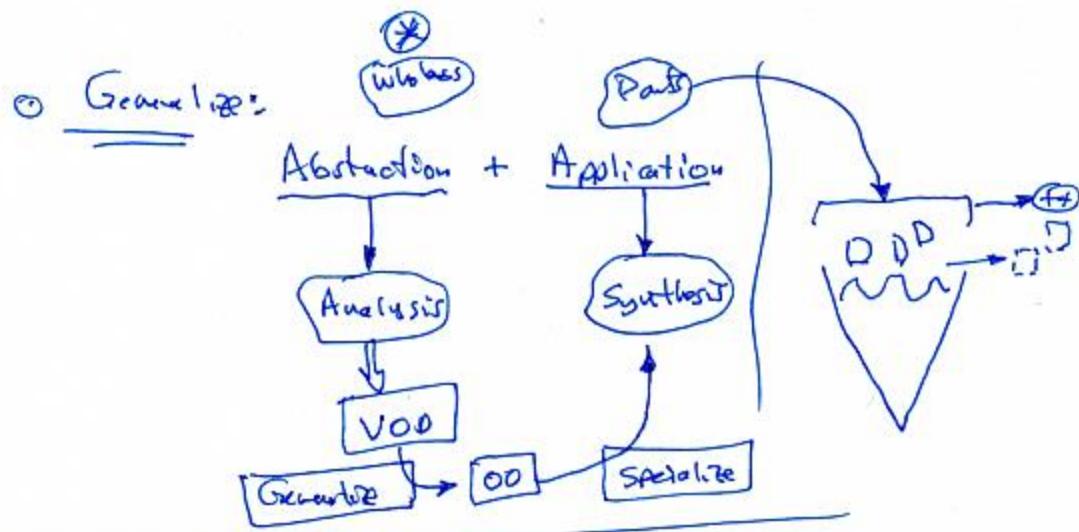
Day 13: Dynamic Proxy, and Factory Pattern

Daily Topics

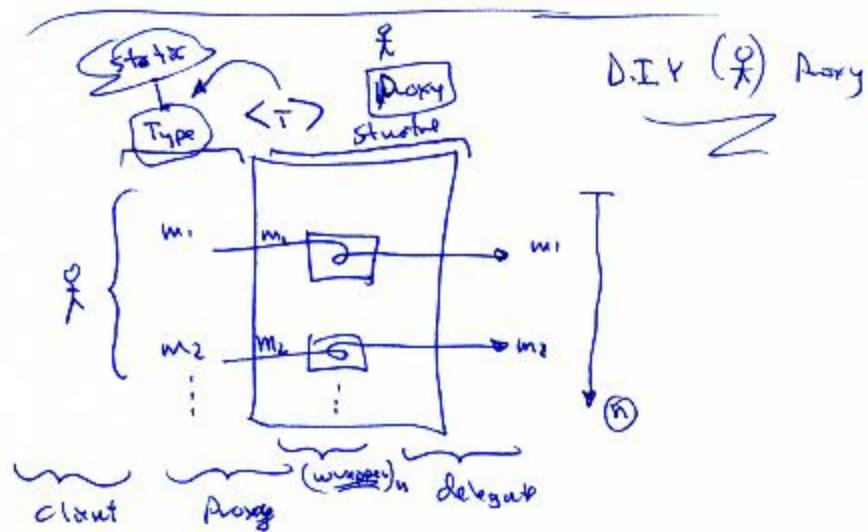
Day 13:

Factory Patterns

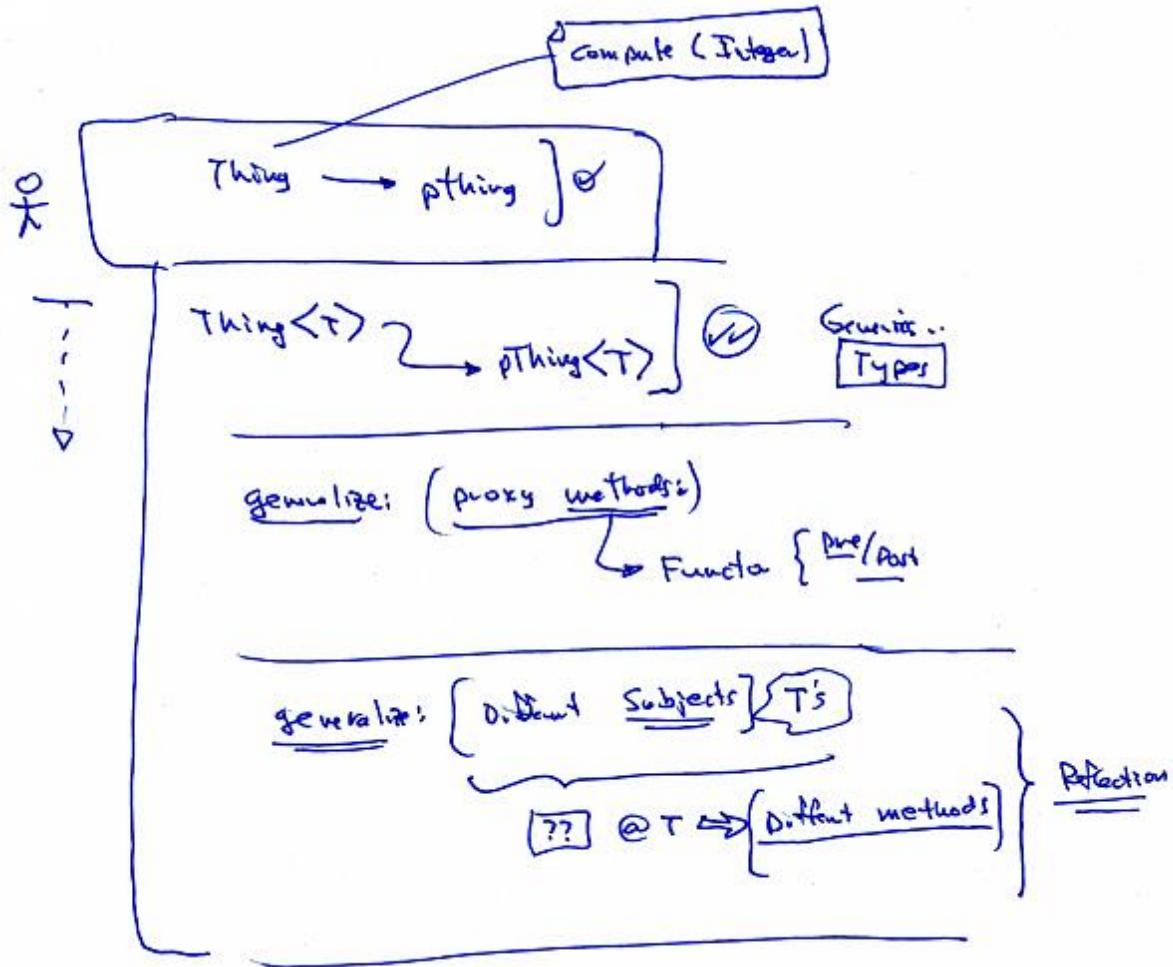
- Review ↗
 - Factory pattern family
 - Simple Factory, (u/I) ← functional (Static)
 - Factory Method, " ← oo (method) / poly... " ← FM
 - Abstract Factory " ←
 - Generic Factories? ① Java
 - Reflective Factory ② → (ERT)
 - ④ No magic, just OQ ↗ (+ Design)
 - Singleton Pattern (Builder, Prototype)
- (ctors <= static) ☺
- ④ Factories + P/w's



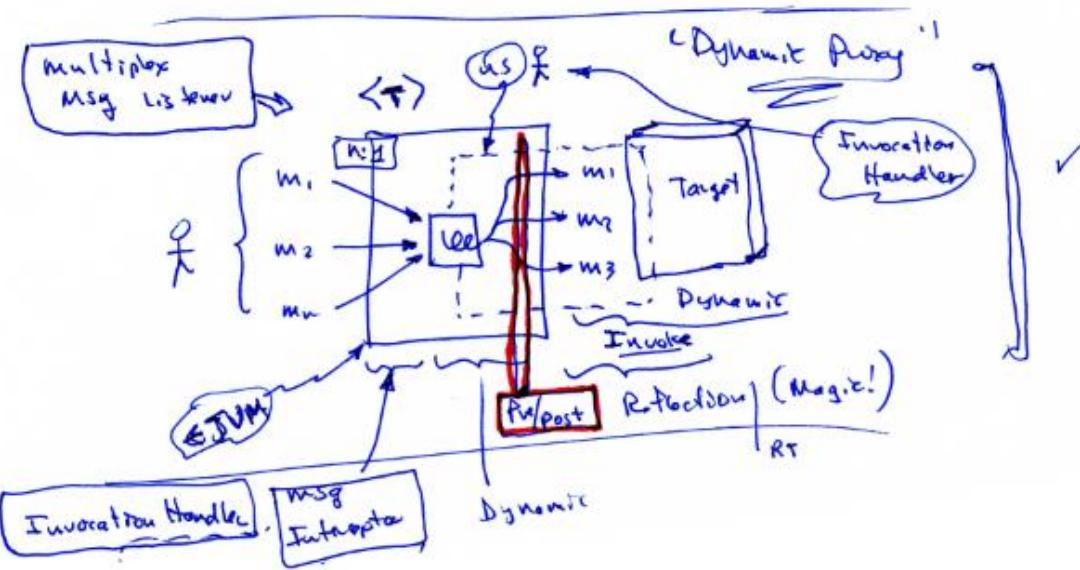
- Generalizations of a pattern and their value



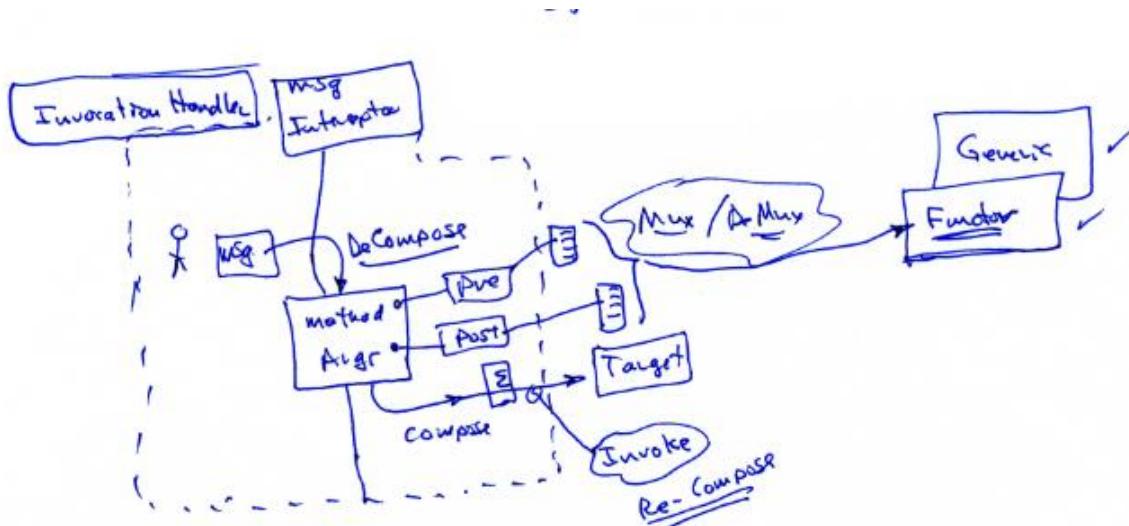
- Traditional (D.I.Y.) proxy structure, one wrapper method for every target method
- Structure of proxy \equiv structure of target



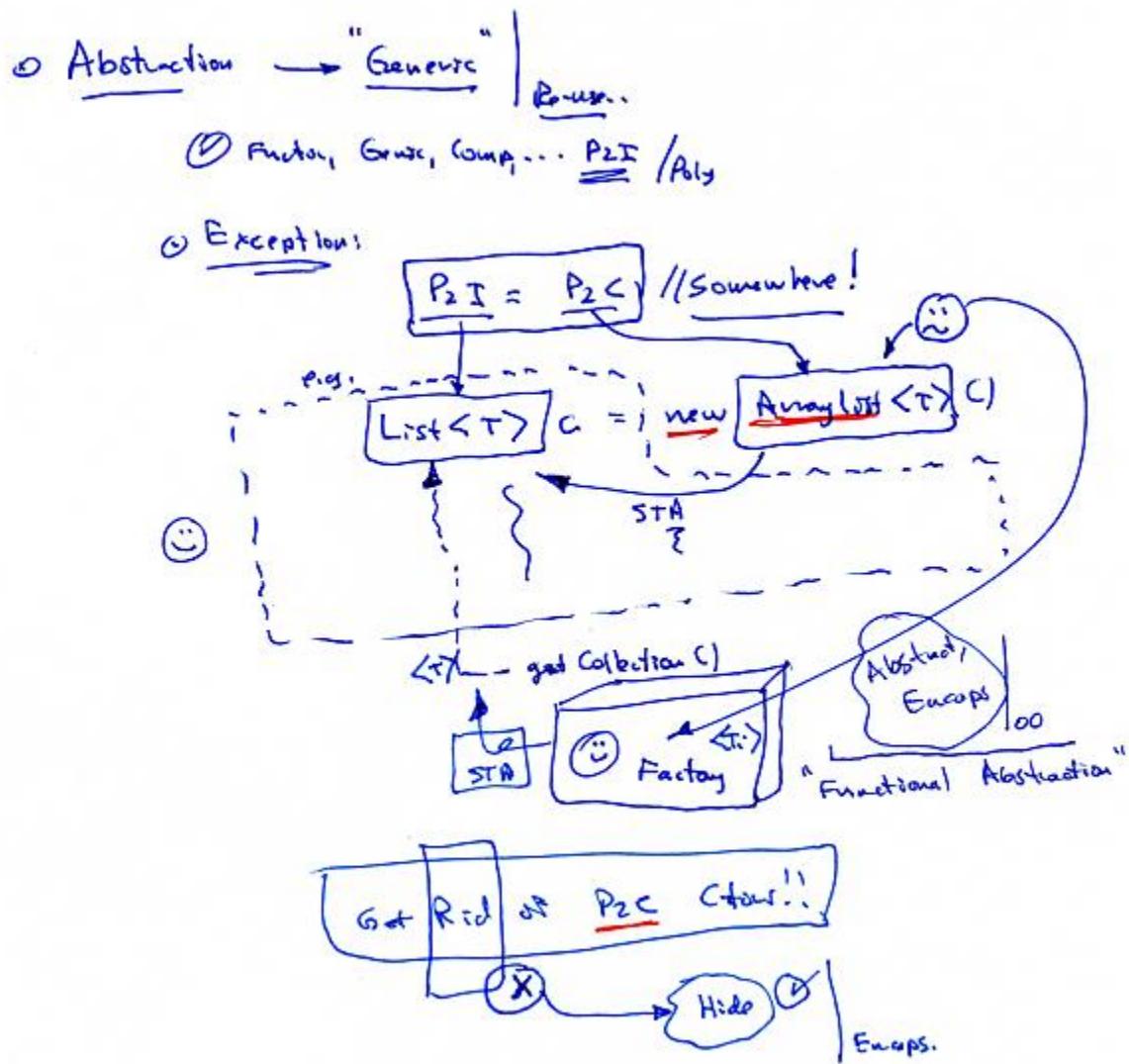
- Generalizations of the Proxy pattern



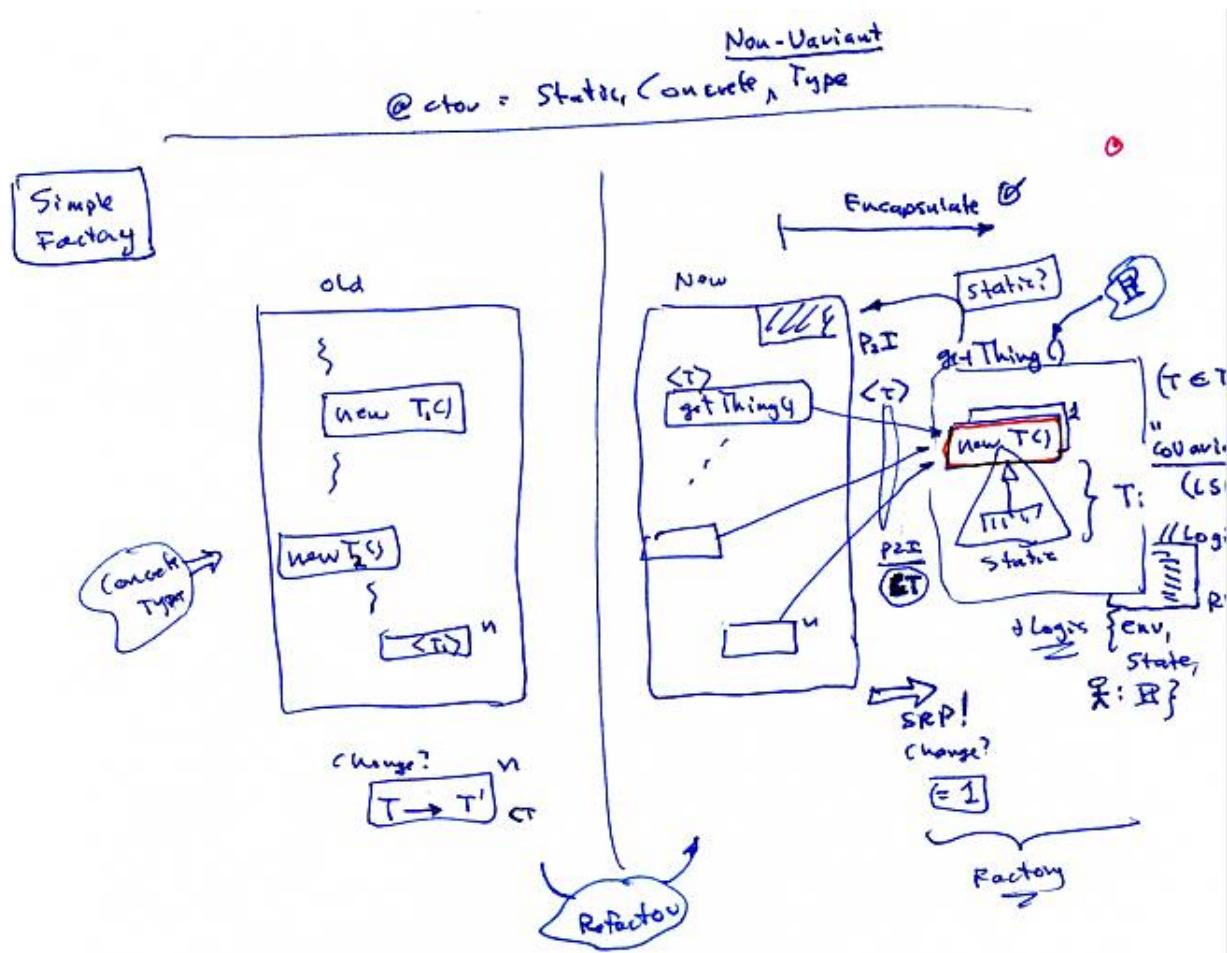
- Structure of Dynamic proxy



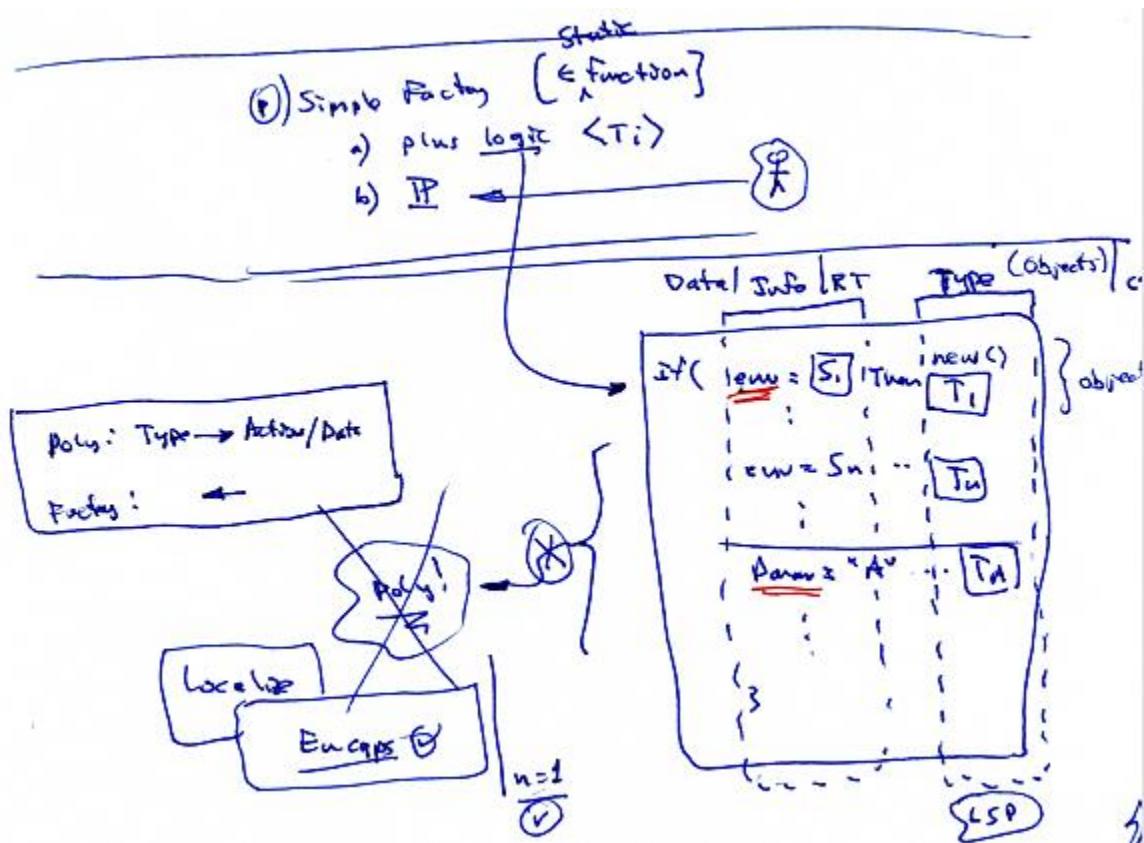
- Usage of dynamic proxy

Factory Patterns ::

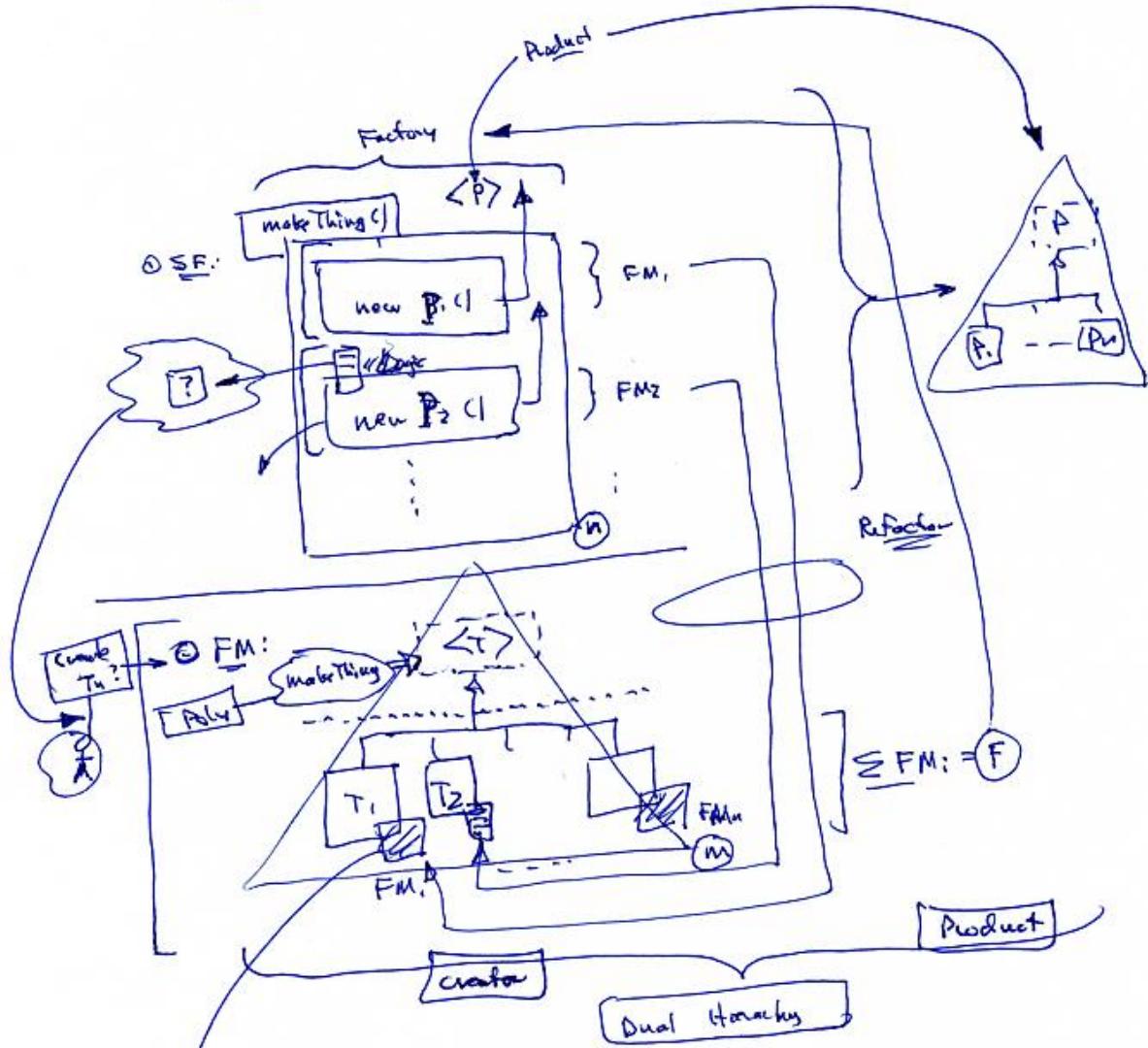
- Abstracting concrete types (*constructors*) from code



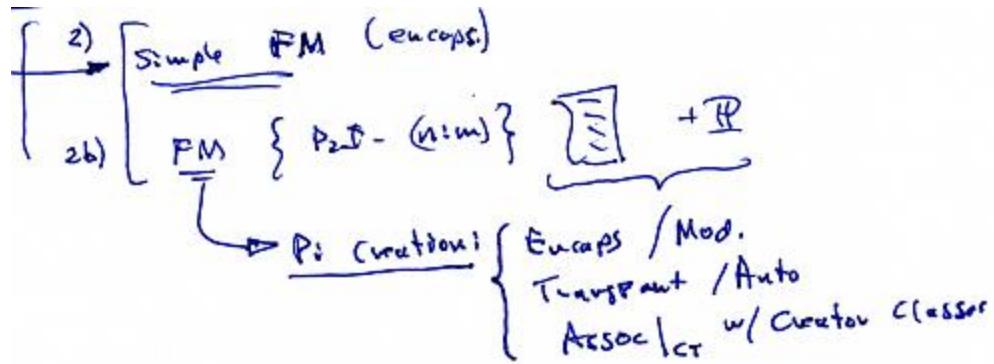
- Phase-1; Simple Factory
- Refactor concrete constructors into a single factory function



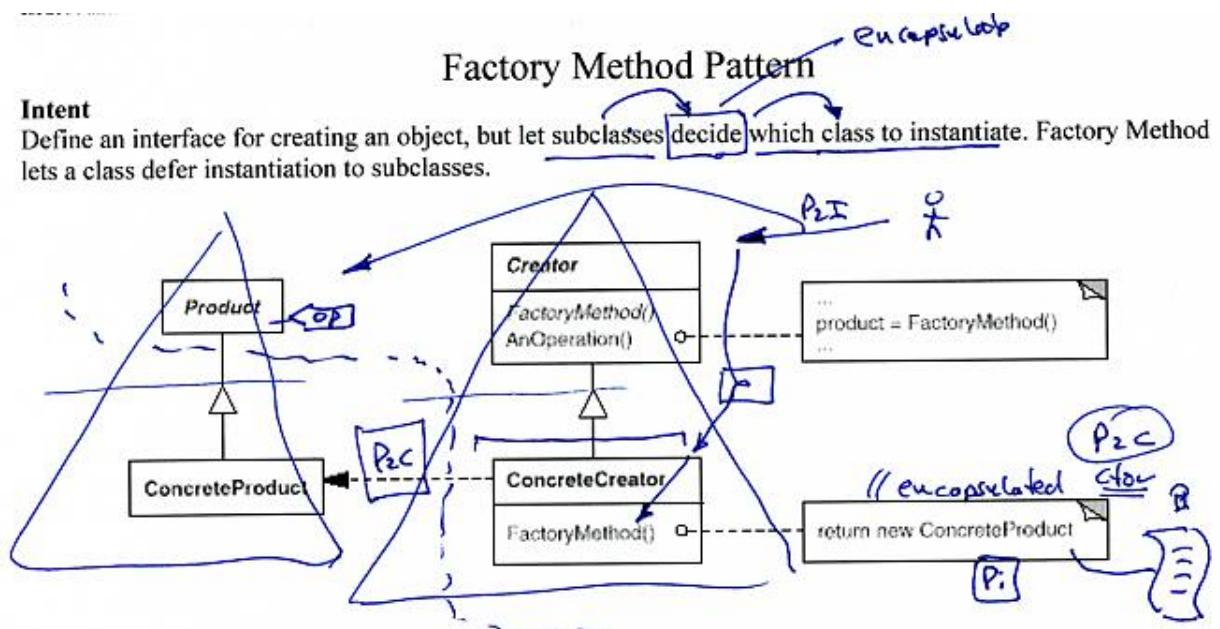
- Could also have a parameter, to allow logic for sub-type selection inside of the factory function



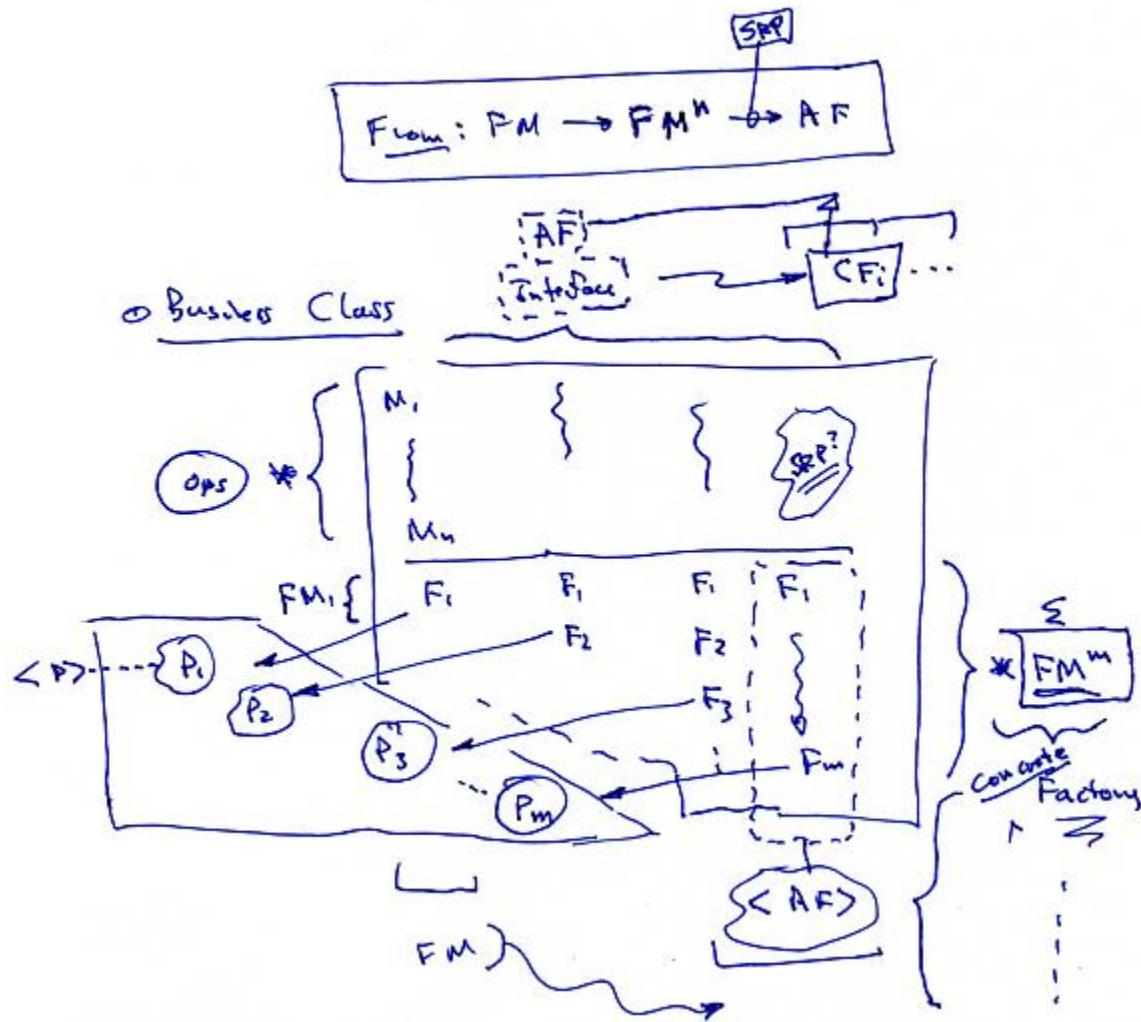
- From a Simple-Factory (function) to a factory Method;
- Refactor code for each concrete type creation into sub-classes of another hierarchy



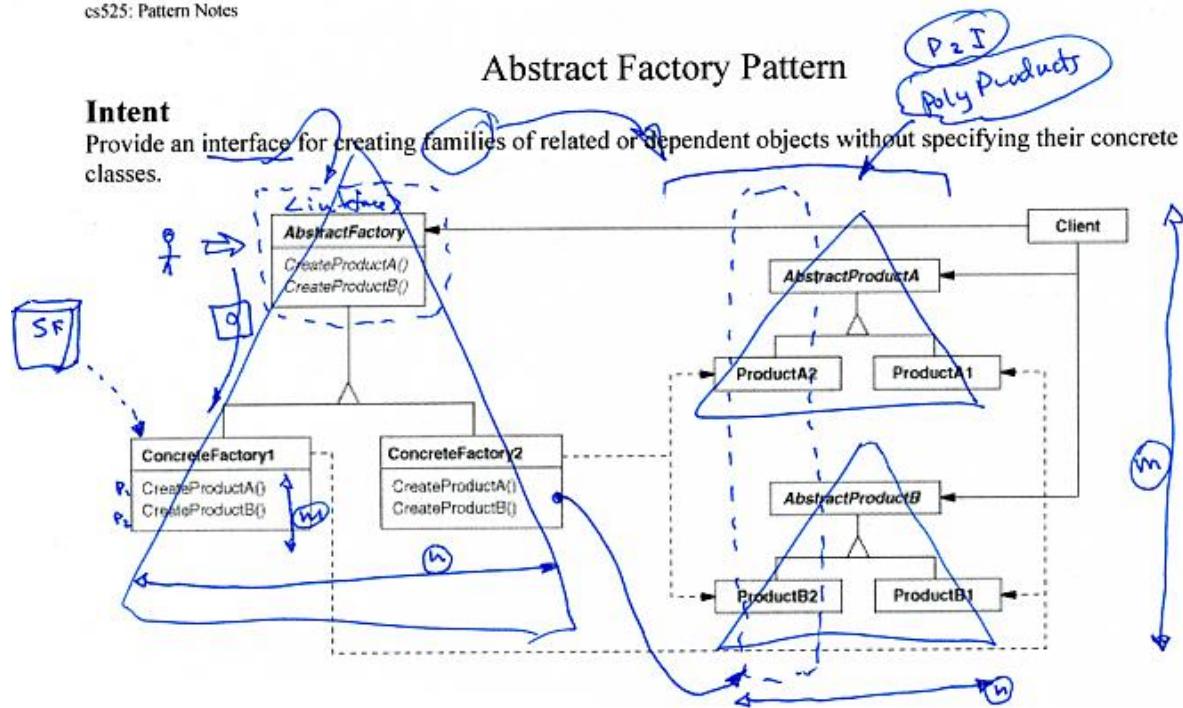
- This is our second class of factory's



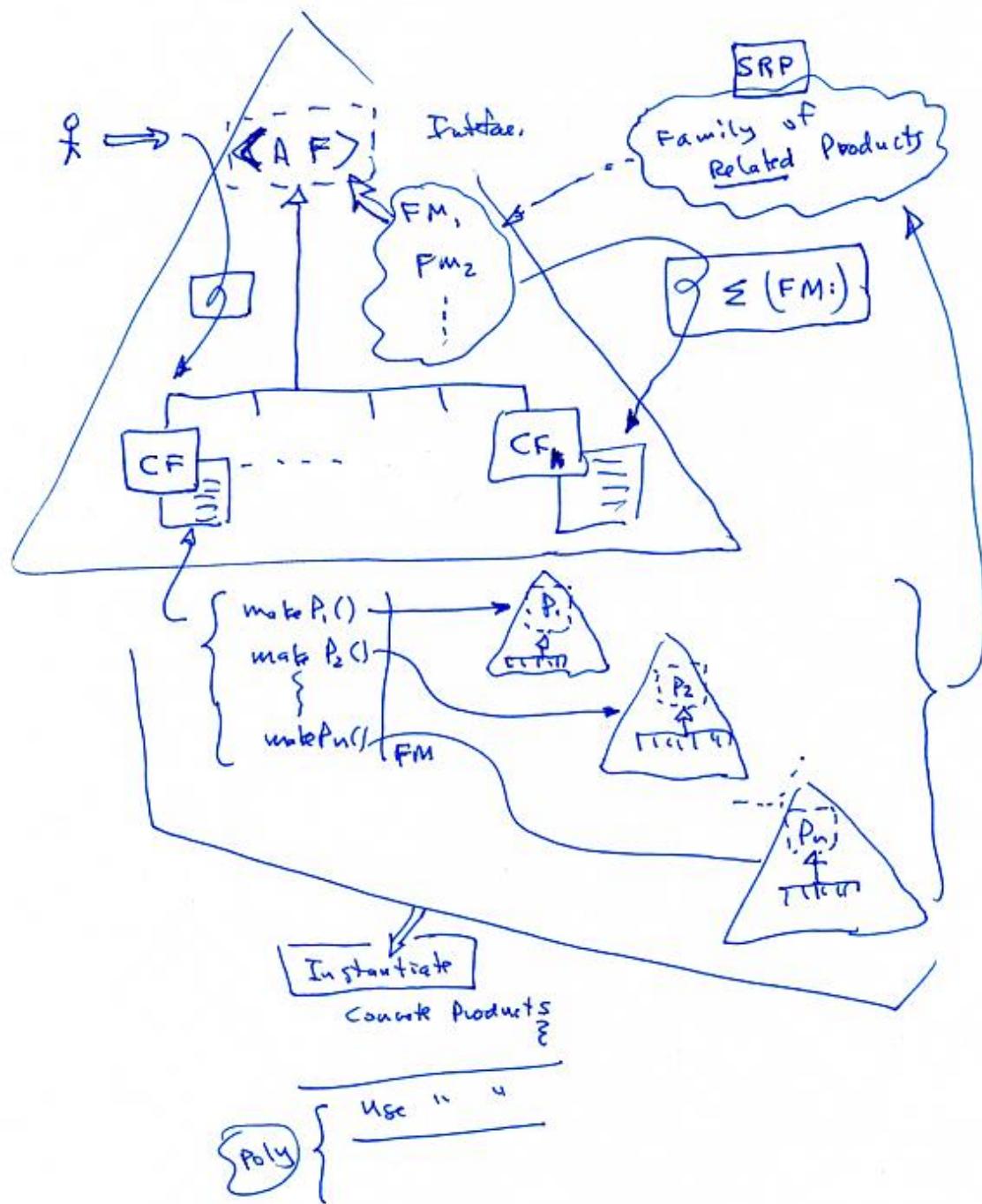
- GOF description



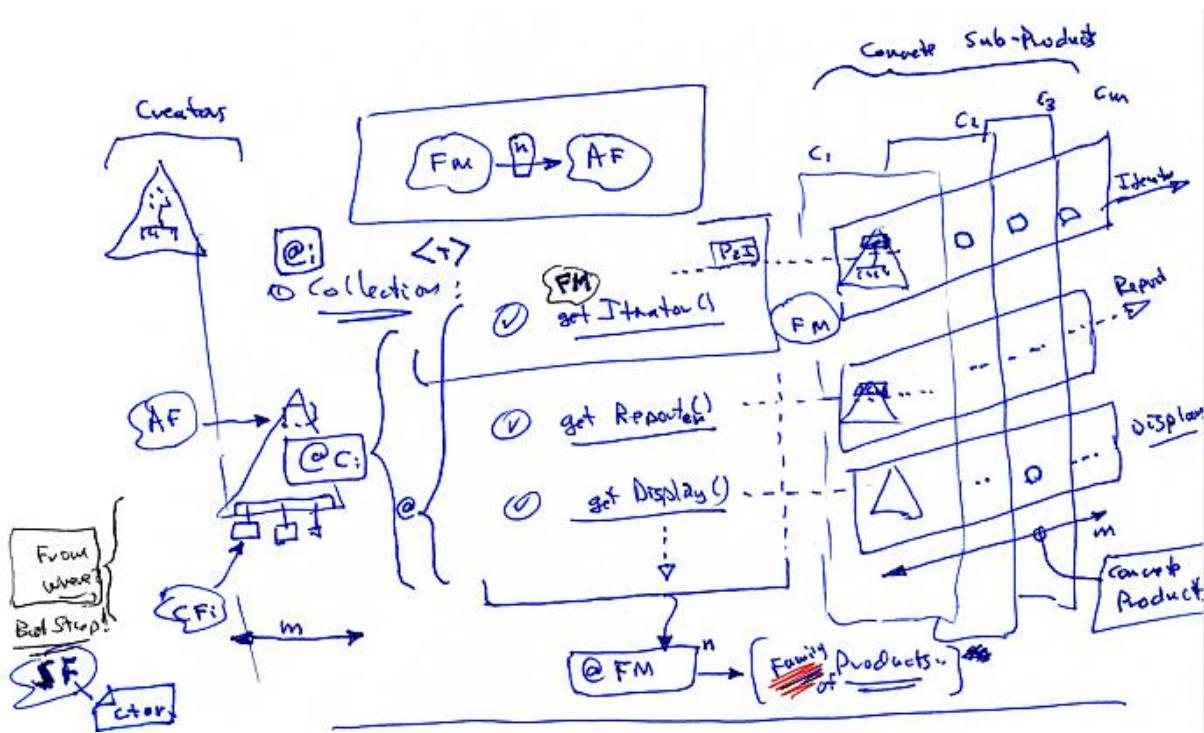
- Generalizing from $FM \rightarrow AF$,
- $AF \approx FM^n$



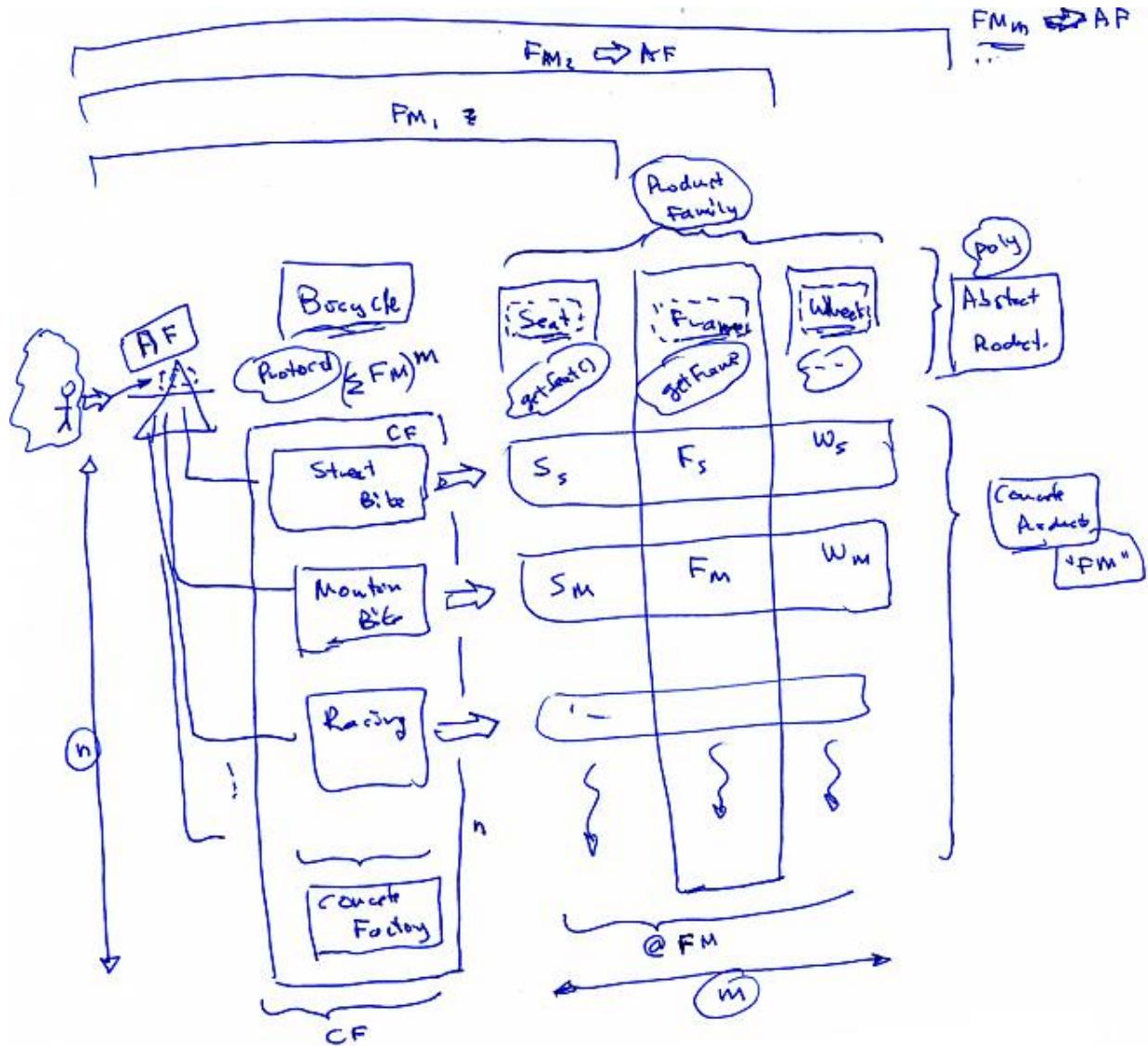
- GOF description



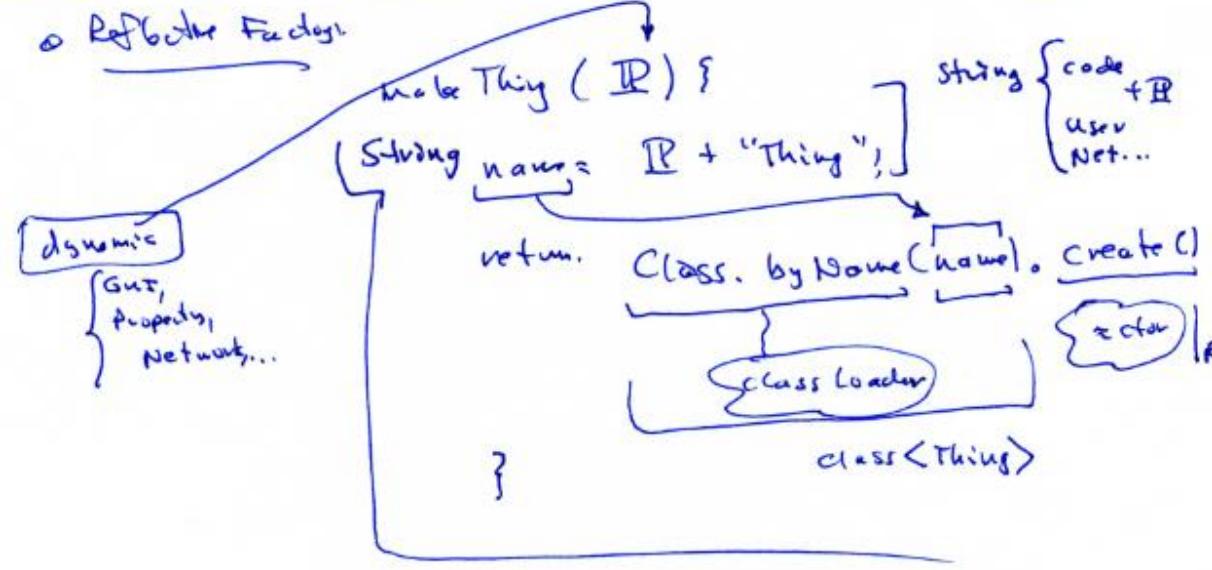
- Structure of AF pattern & usage



- Example of using AF pattern

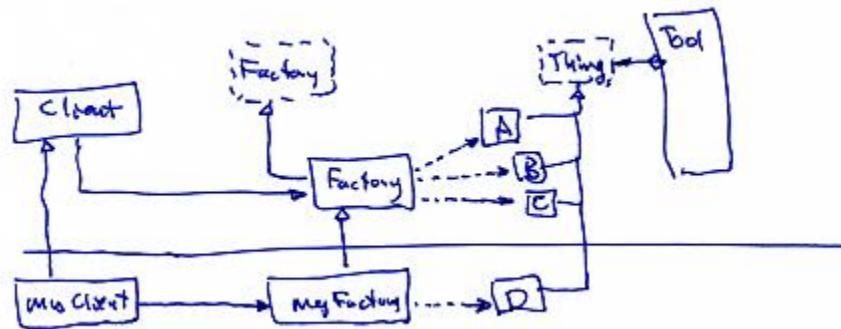


- Another AF example – bicycle shop

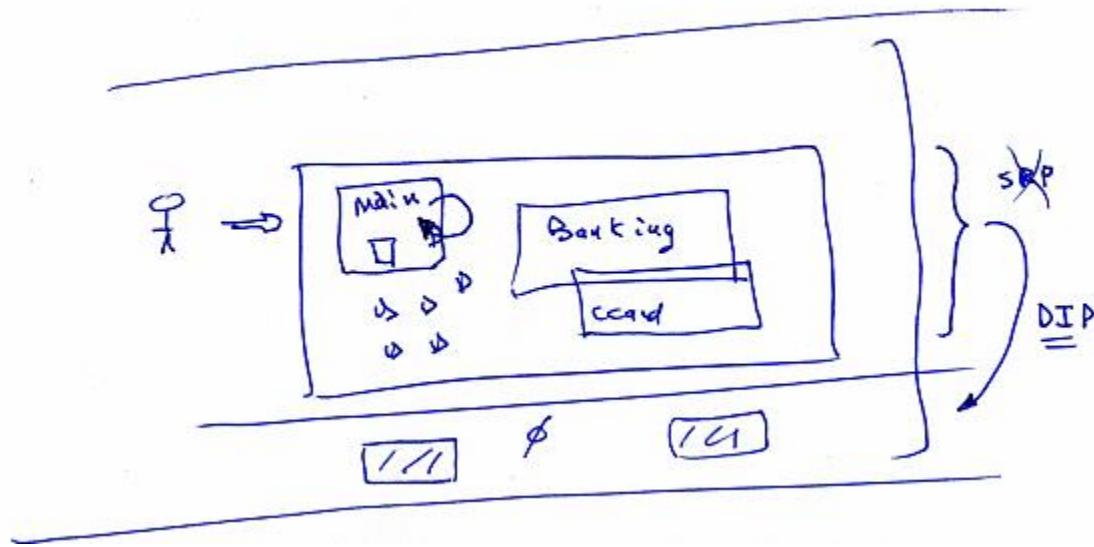


- Reflective Factory

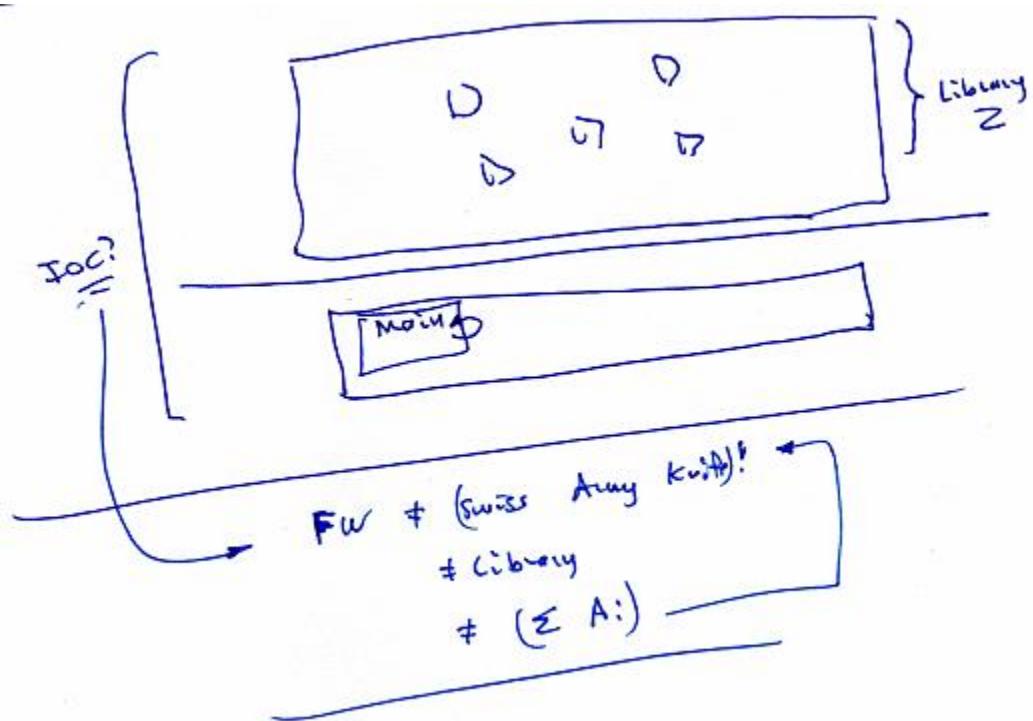
Day 14: Project Assignment



- Using factories with Frameworks

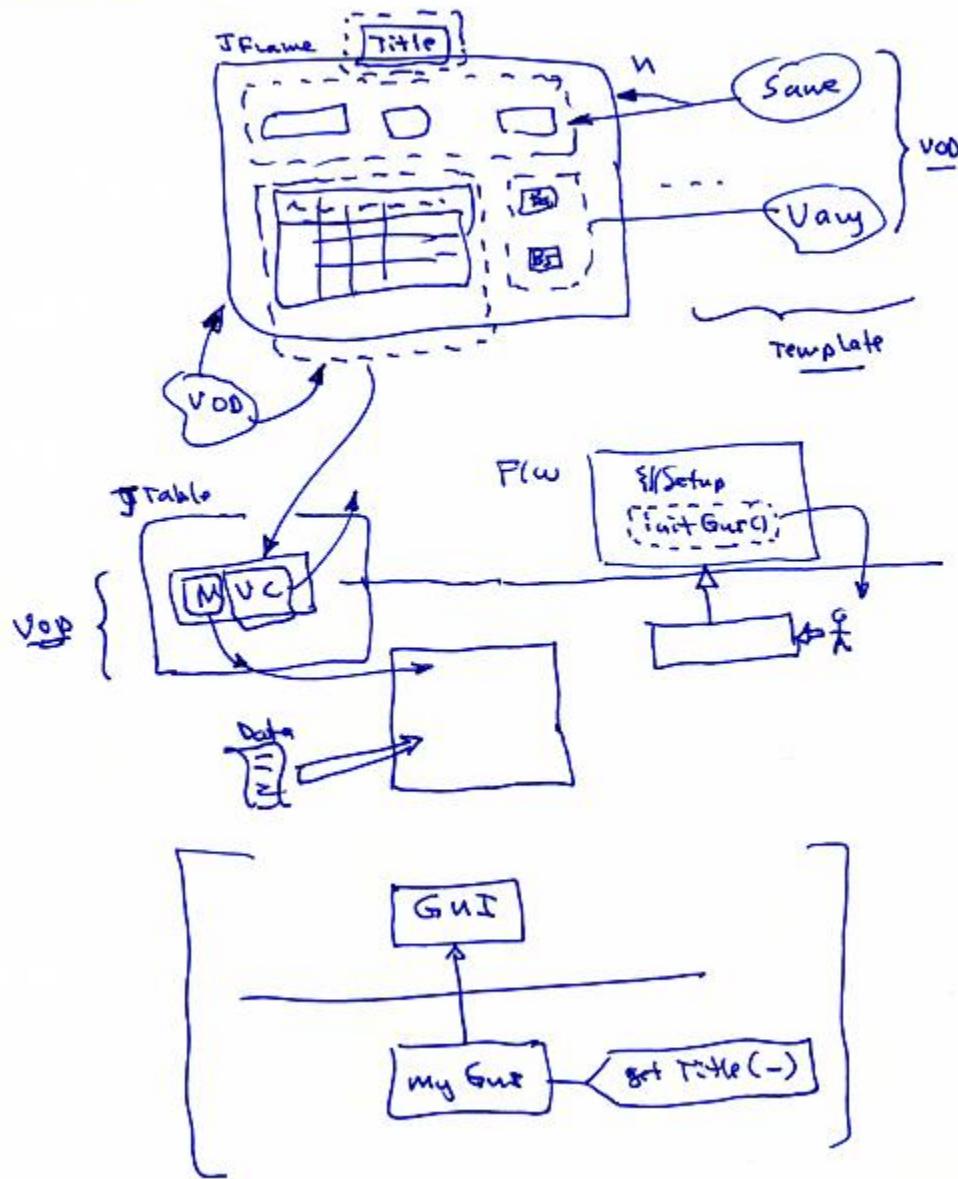


- DIP principle and frameworks



- Framework \neq Library, or combination of applications

o Swing / GUI:



- Refactoring GUI for a framework