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

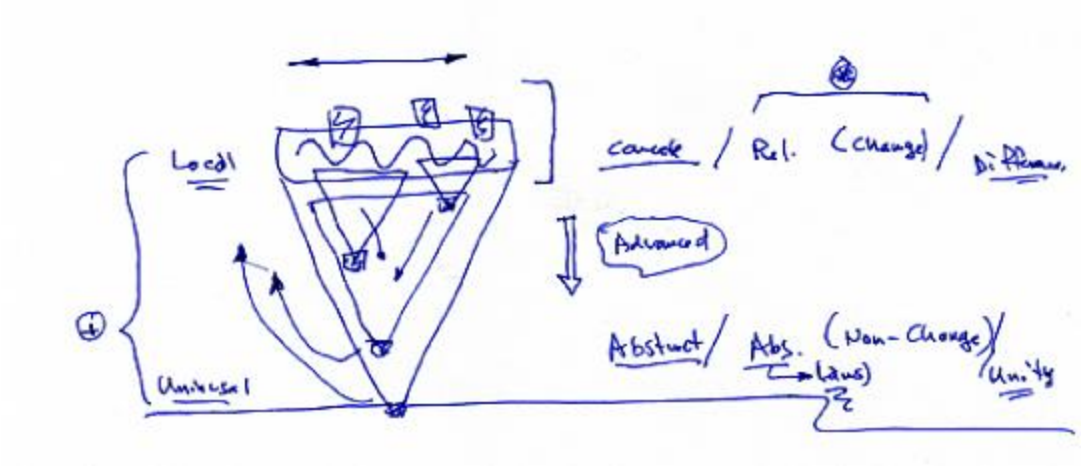
July 2014

Lessons:

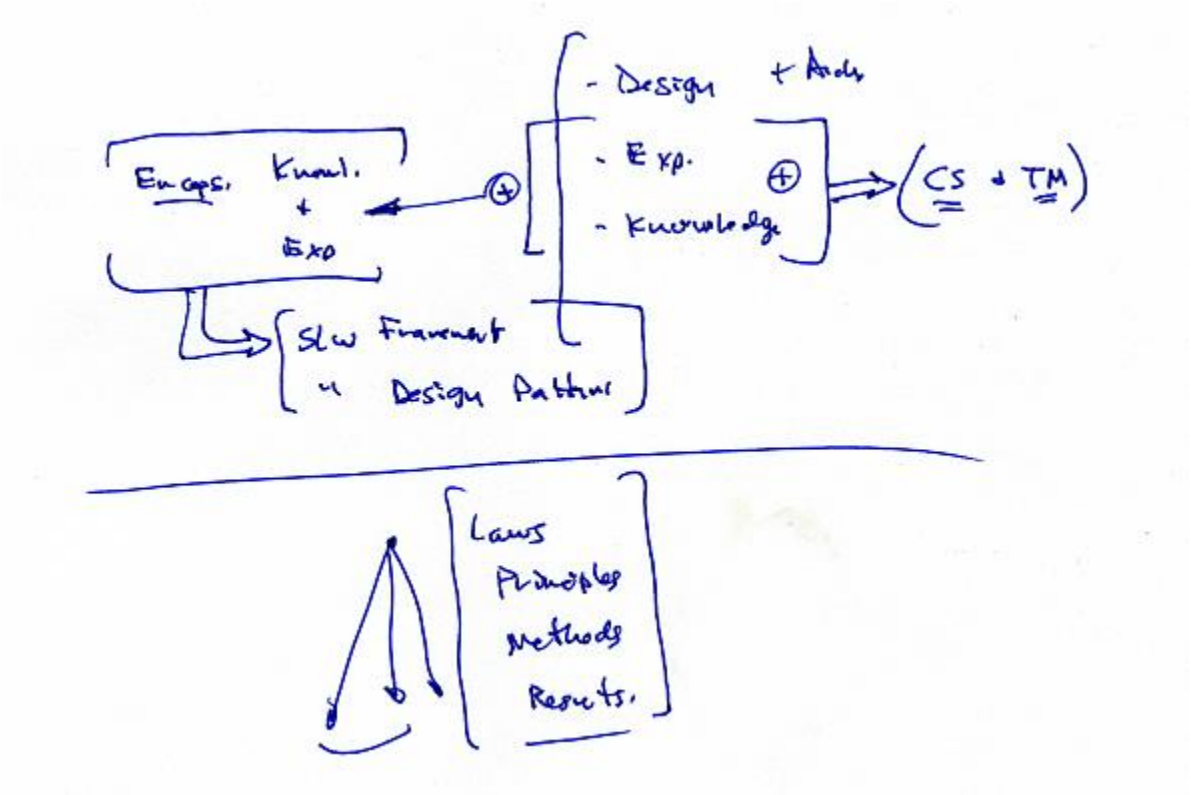
Day 1: Introduction and Overview.

Day 1: Introduction and course Overview

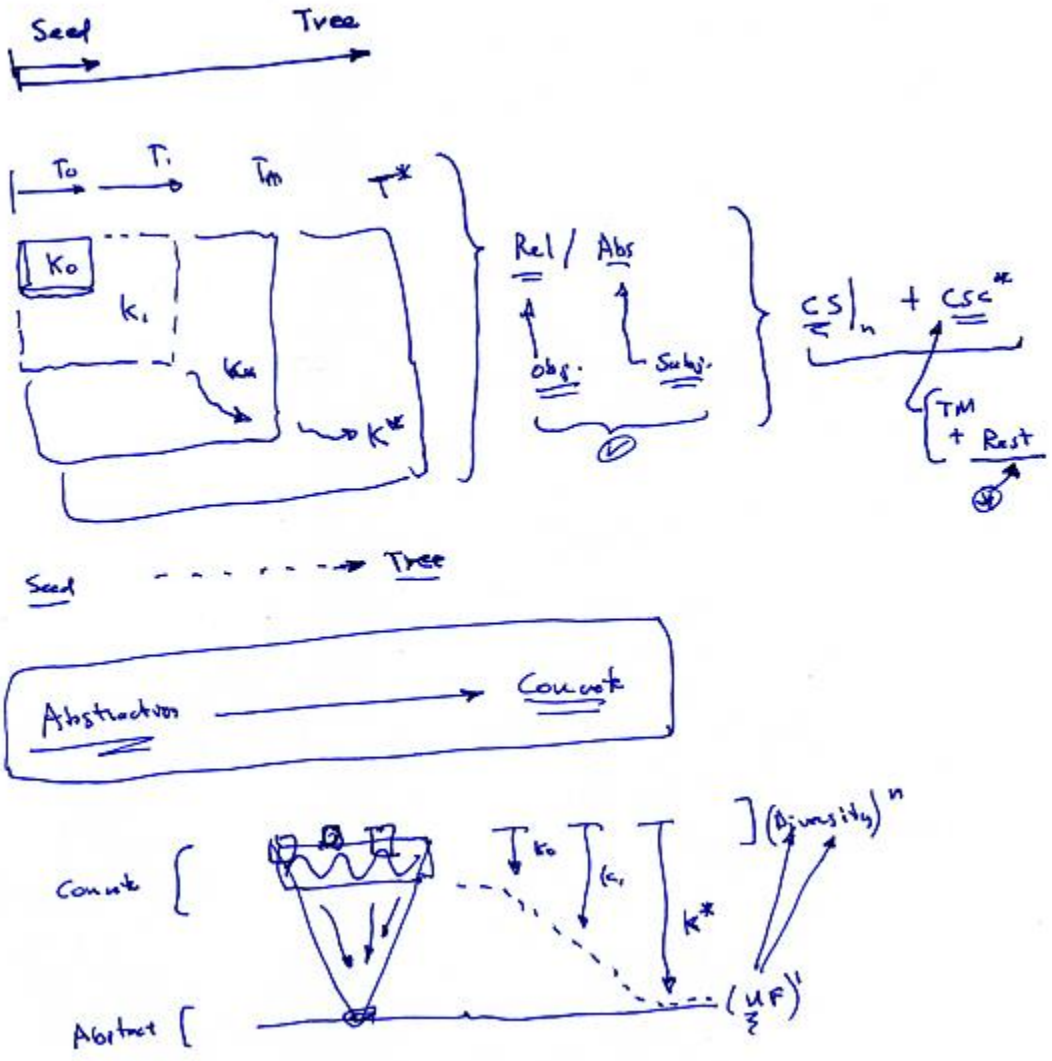
- Overview of Advanced Software Systems
  - Value of Abstraction
  - Elements of Software Development
- Definition of Advanced Software Development & topics
  - Methods of ASD
  - Refactoring and SE++
- Course Structure & Administration
  - Daily Schedule
  - Quizzes
  - Labs:
    - *Timelog*, hand-in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



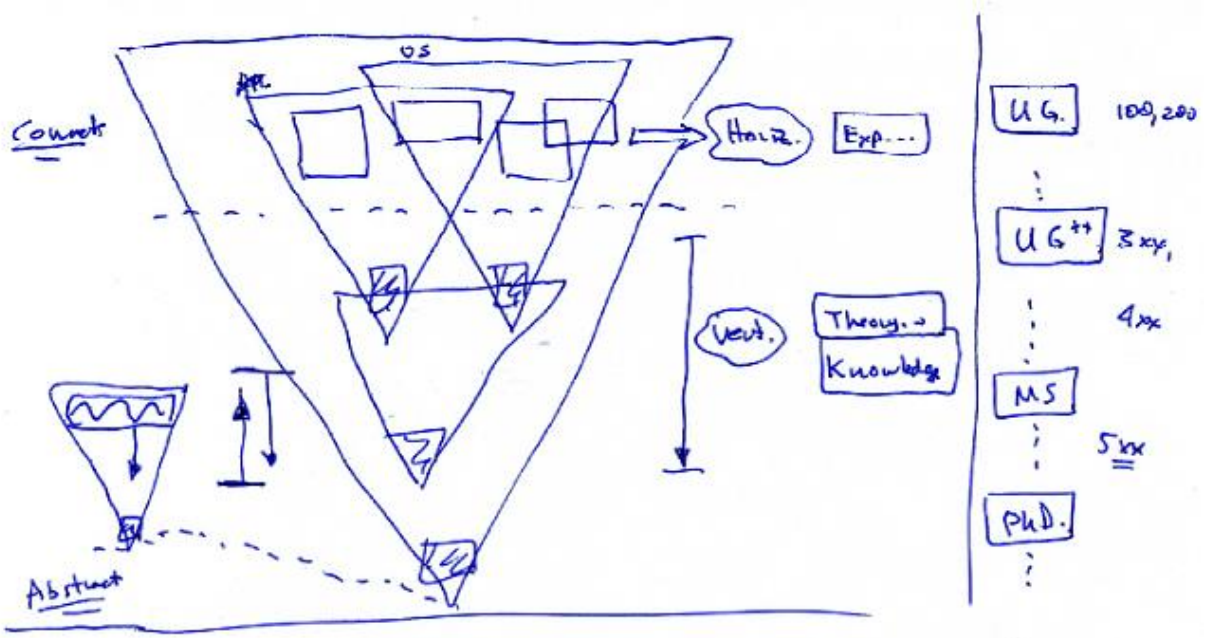
- Levels of abstraction

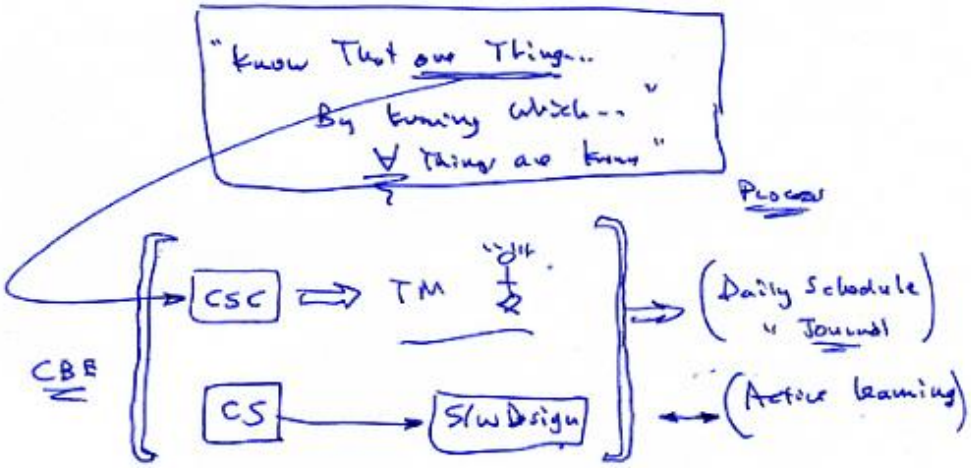


- The meaning of Advanced Knowledge

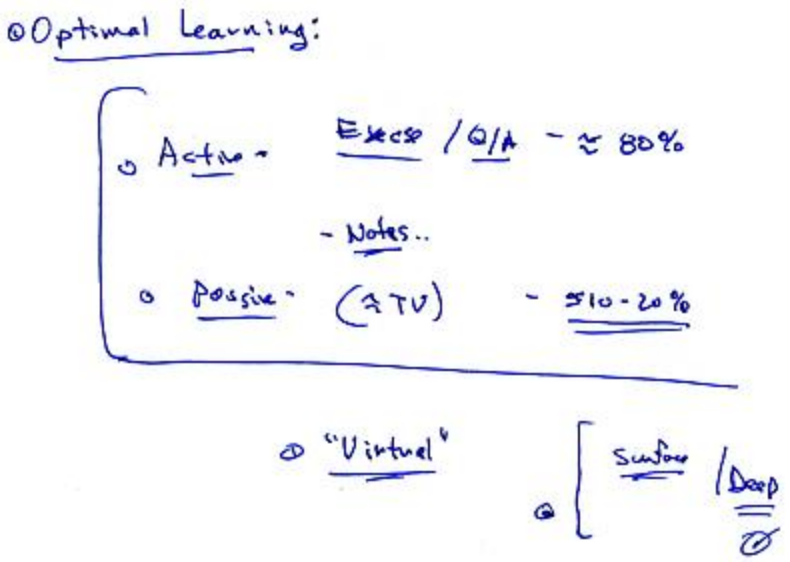


- Levels of Knowledge and Abstraction, from Abstract to concrete, from wholeness to details of expression.

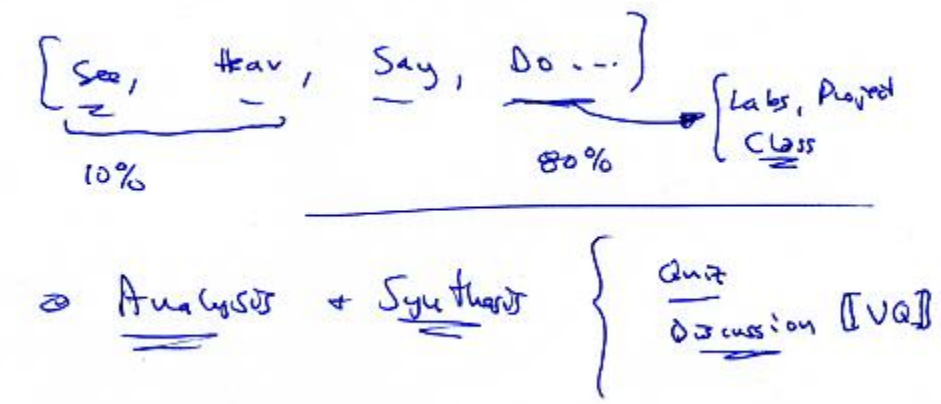




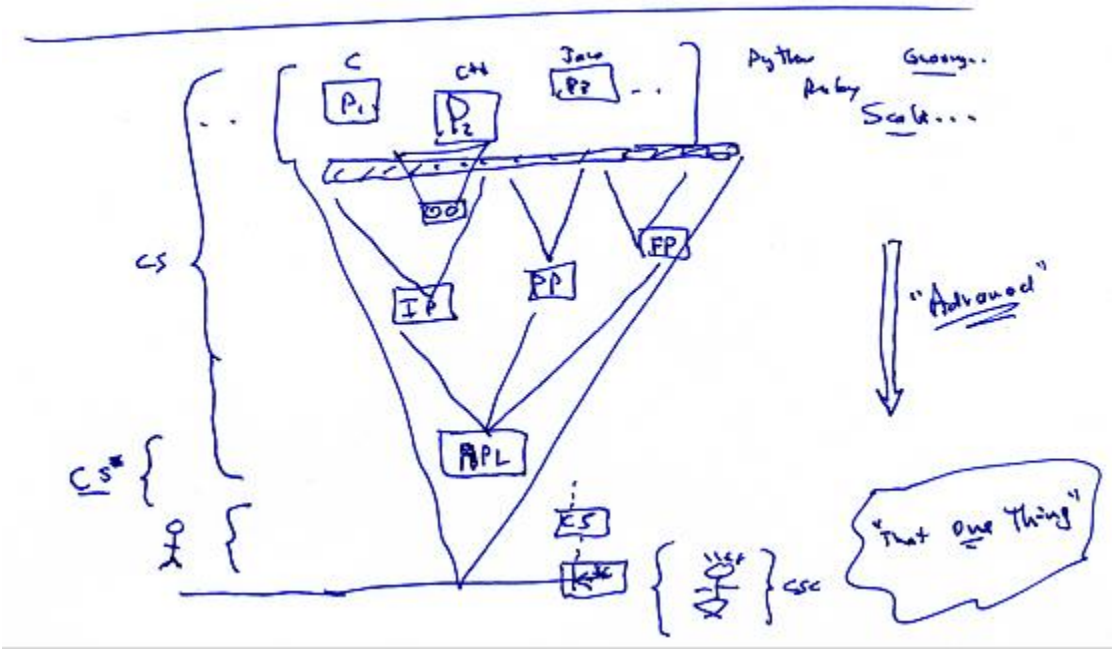
- The importance of Consciousness



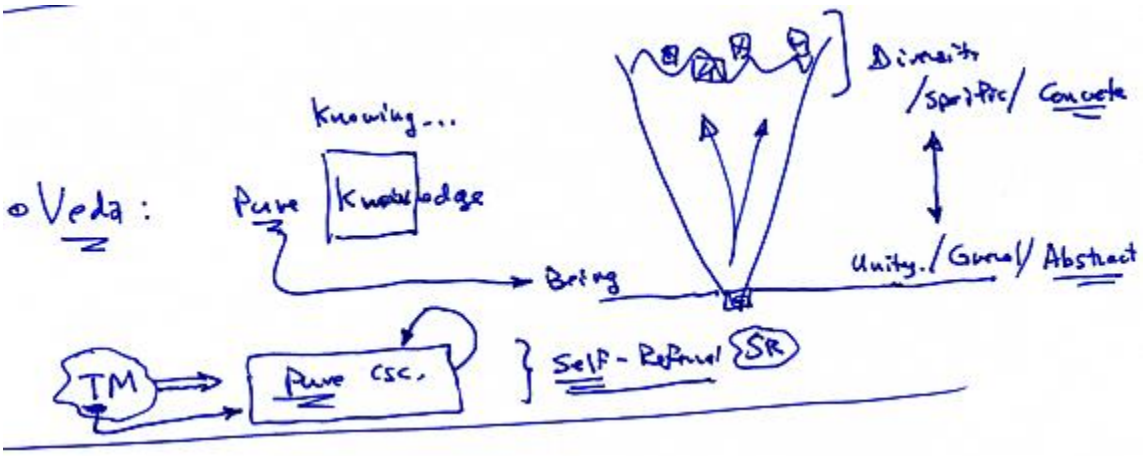
- Active Learning



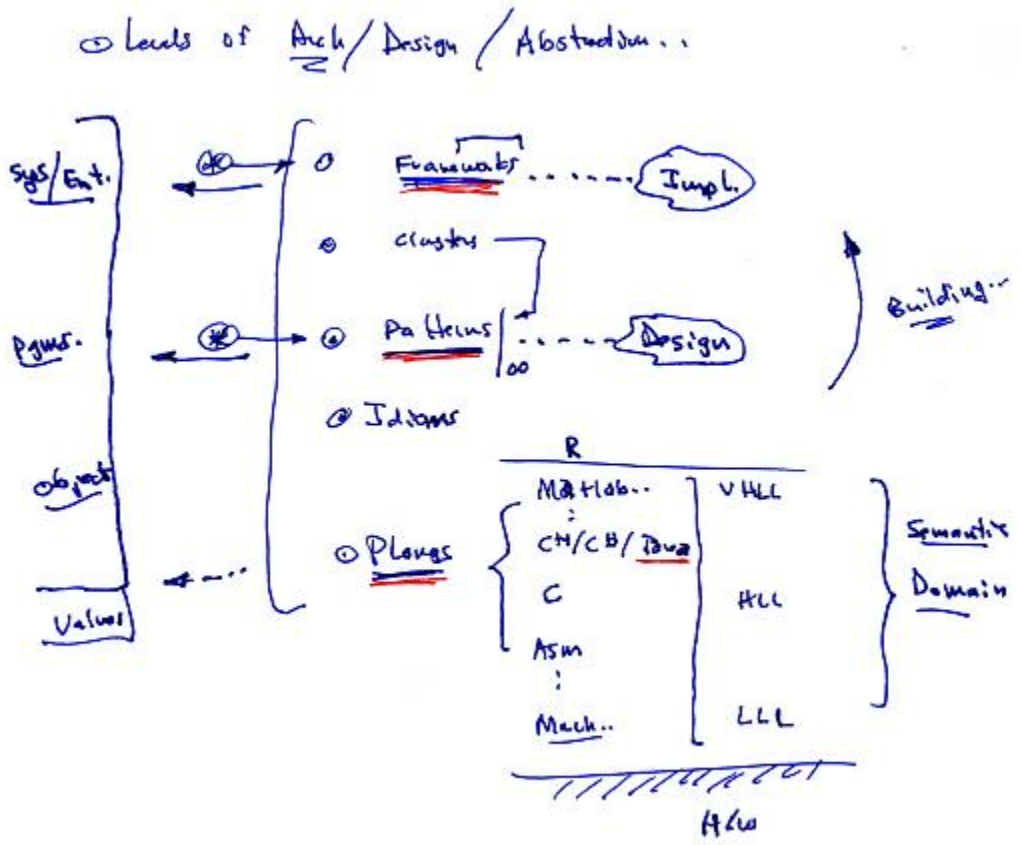
- Active Learning
- The importance of a good daily schedule



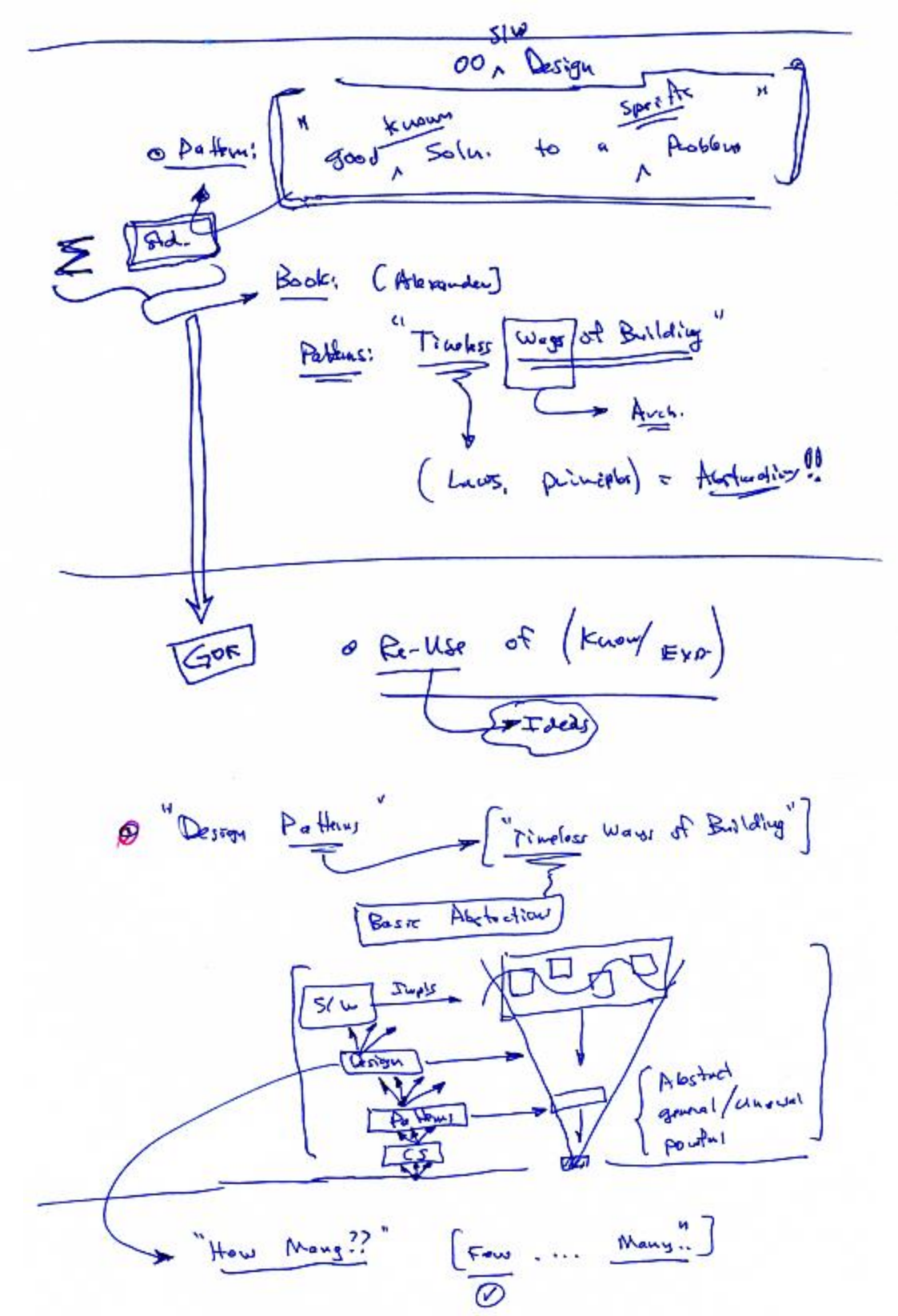
- Levels of Abstraction
- More abstract ⇨ more general, more universal, more powerful



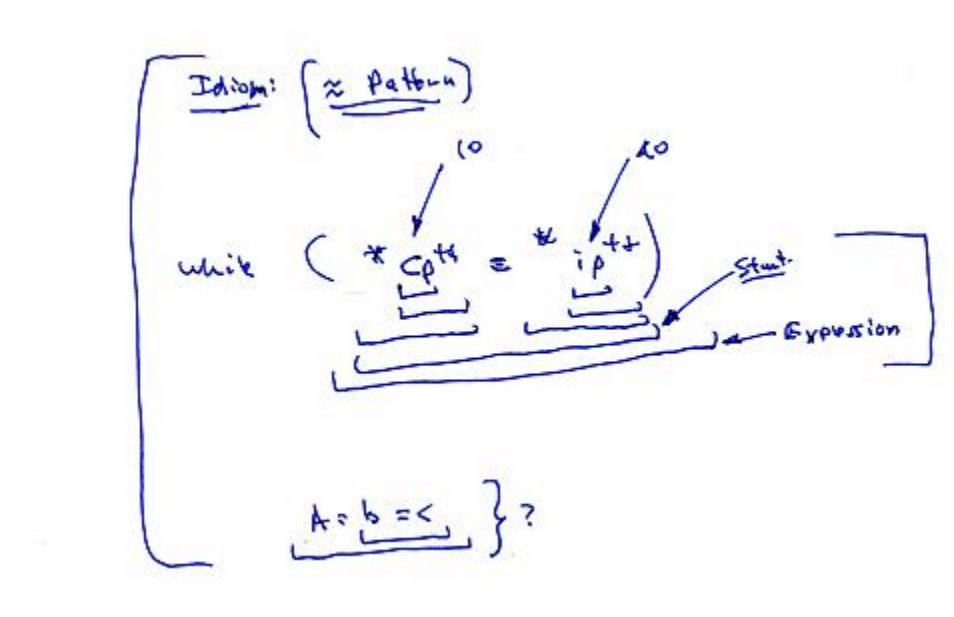
- Levels of Knowledge; Veda ≡ Total Knowledge



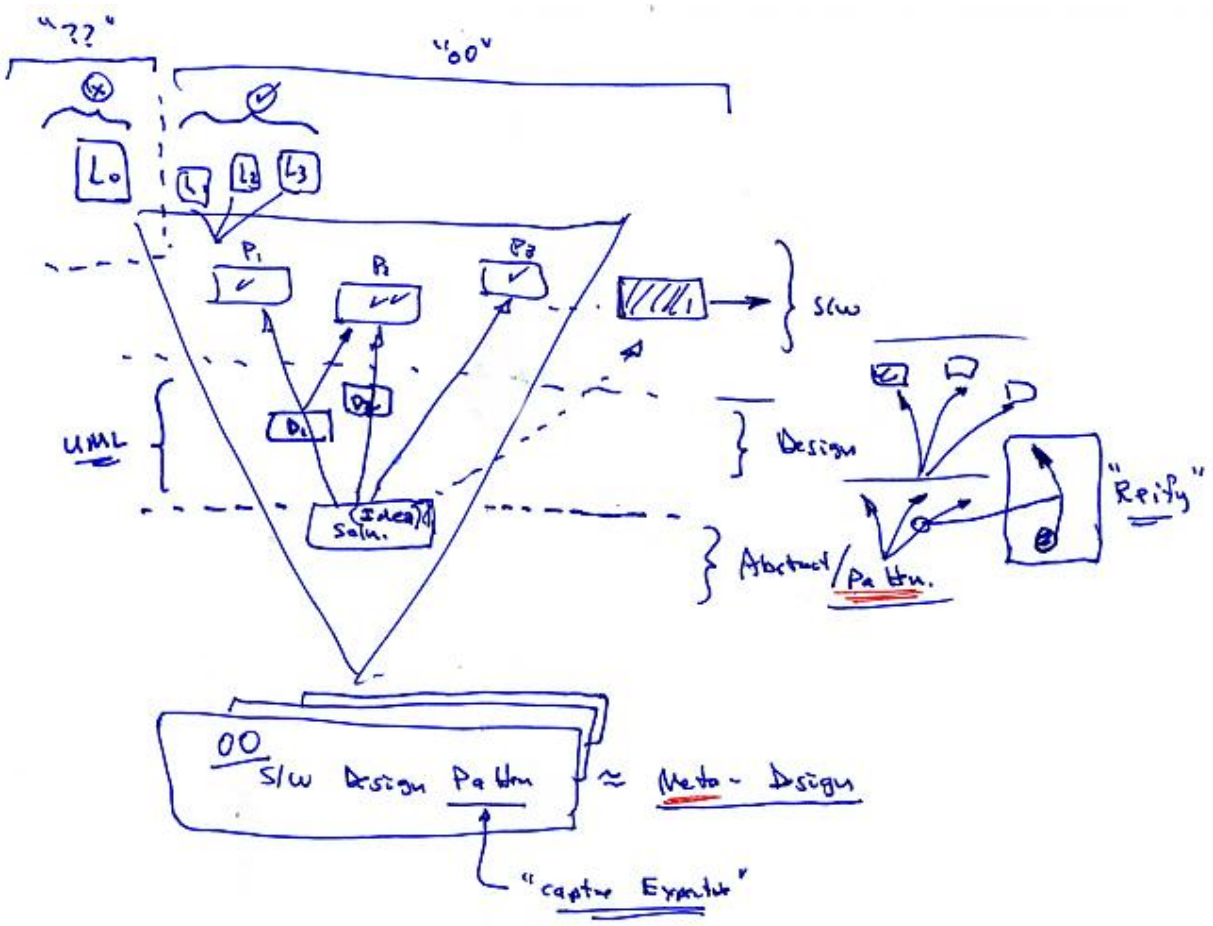
- Levels of Architecture & Design



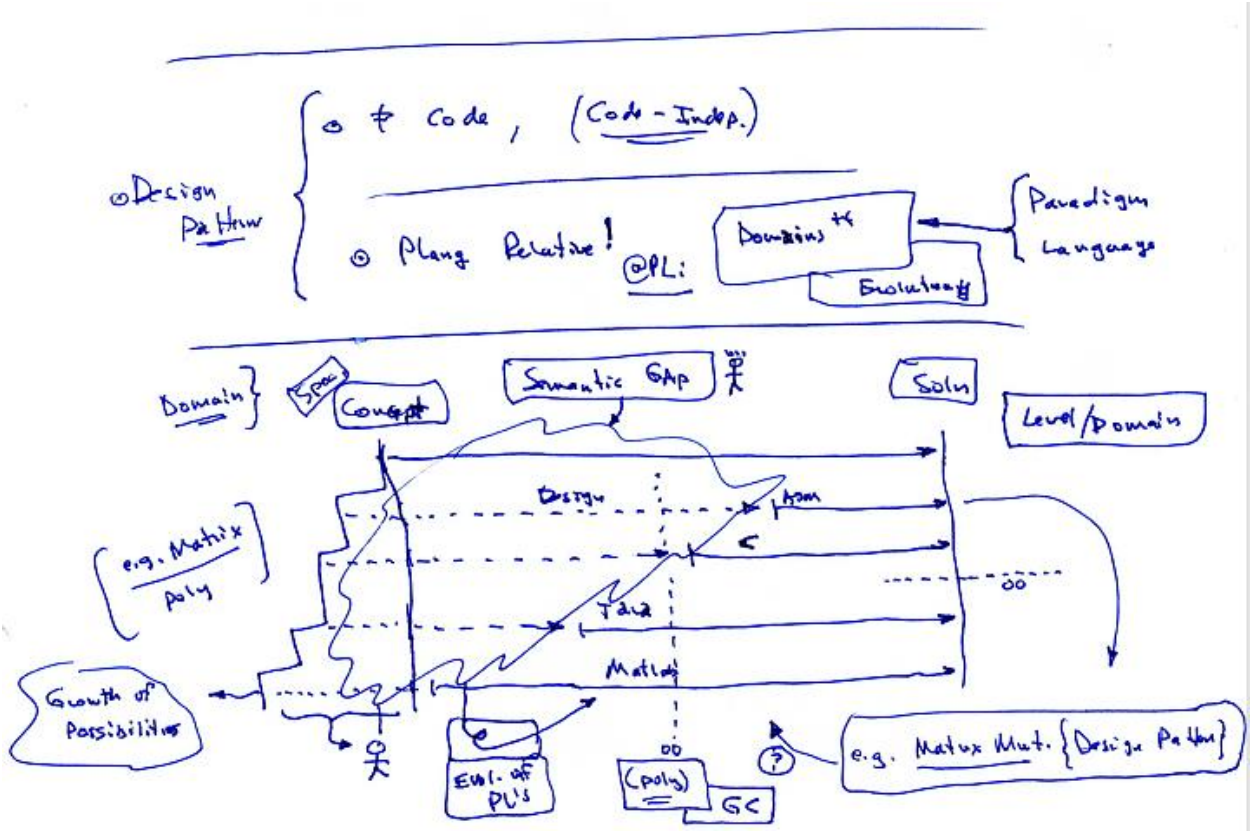
- Definition of Patterns



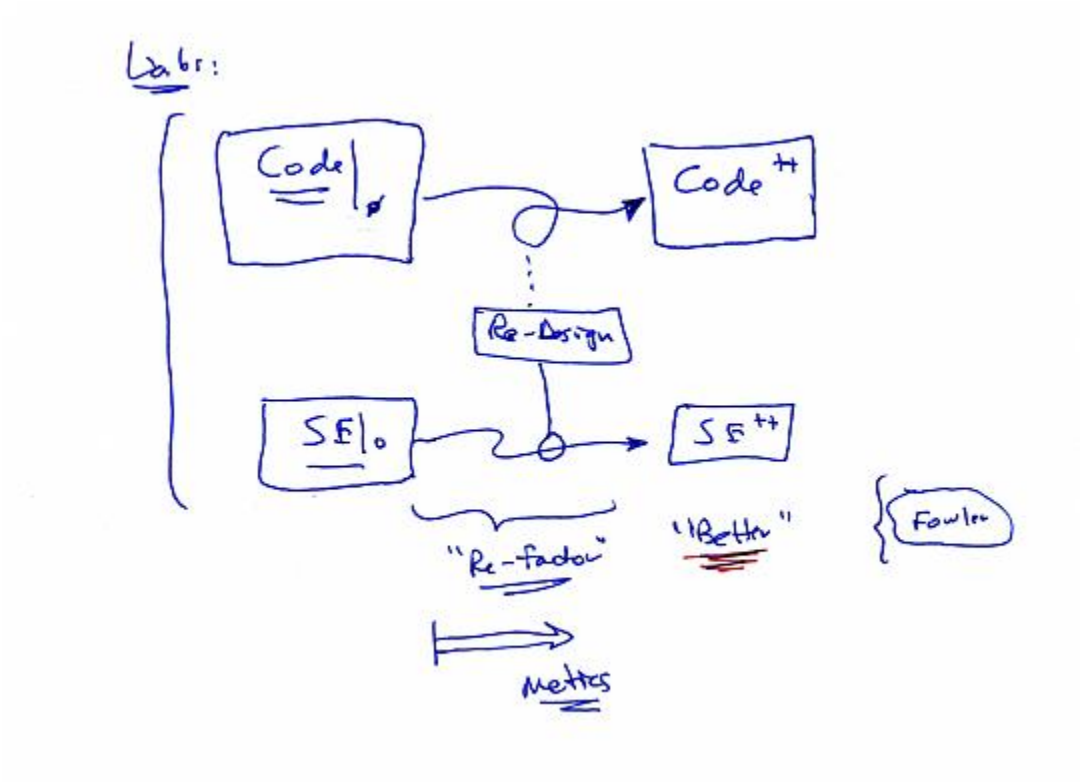
- Idiom = Mini-patterns

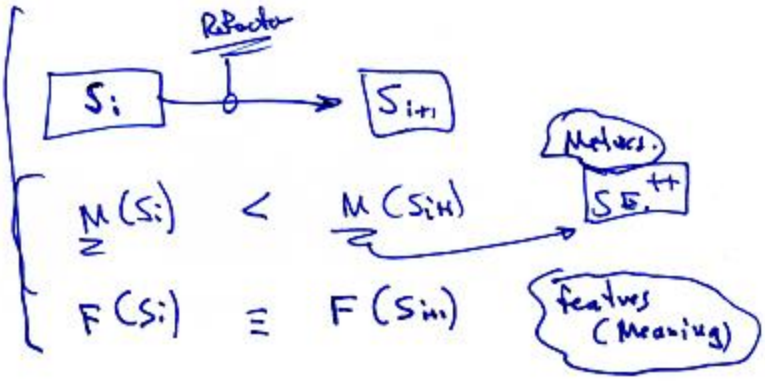


- Patterns are more abstract than designs
- "*Meta-Designs*"
- Patterns *reify* to designs

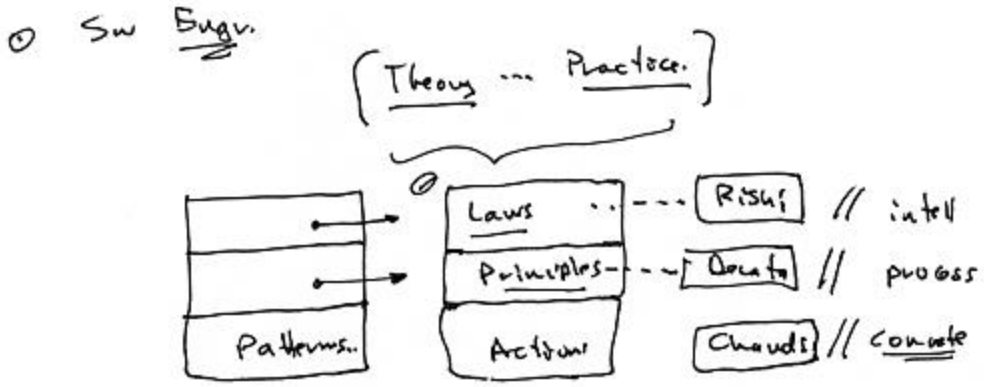


- Relationship of Patterns and Programming languages



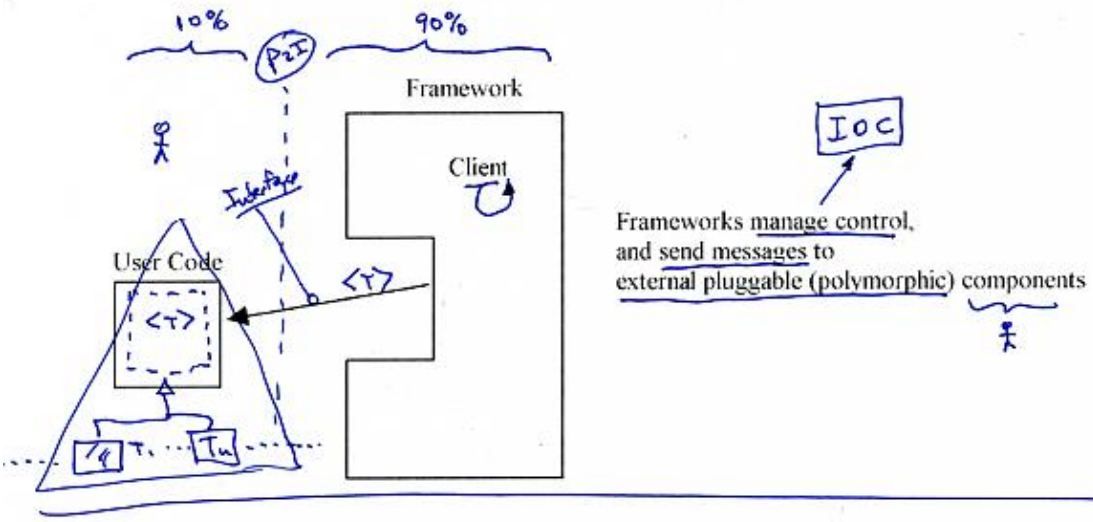


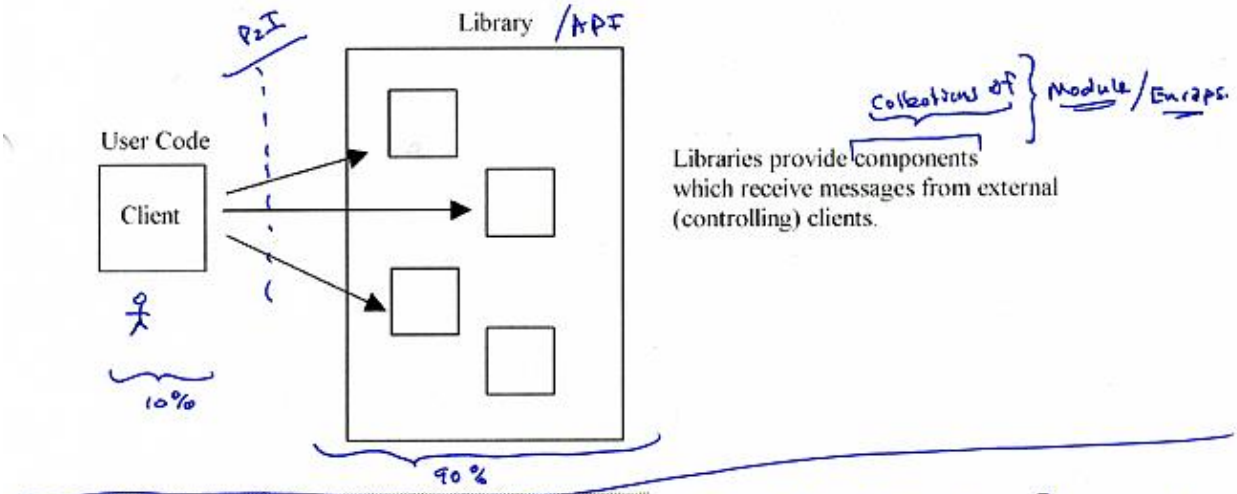
- Refactoring

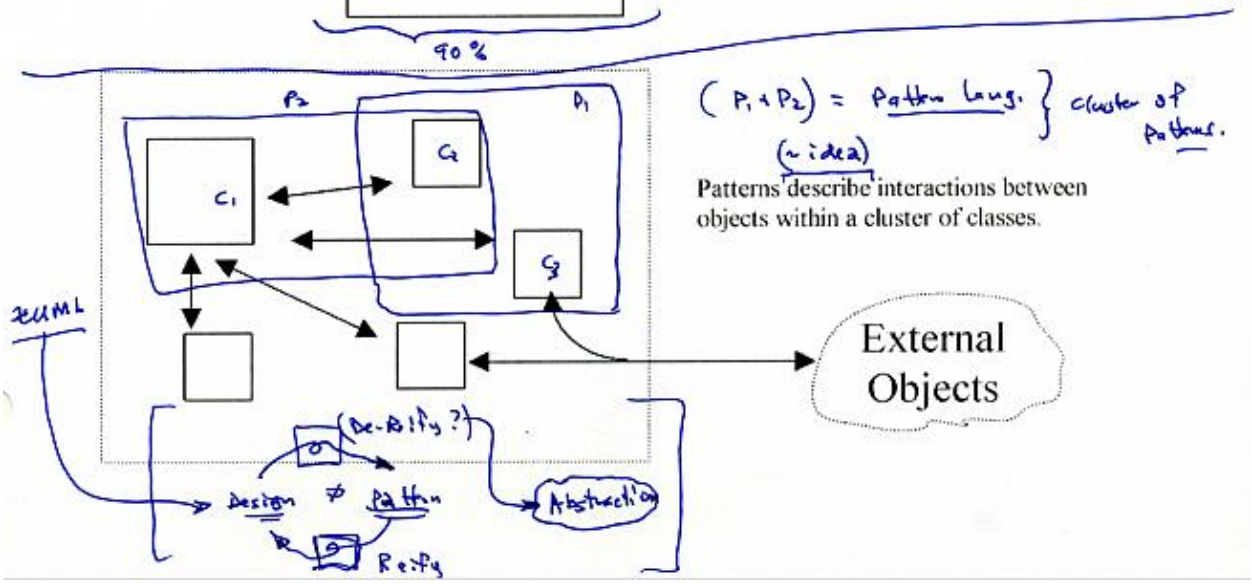


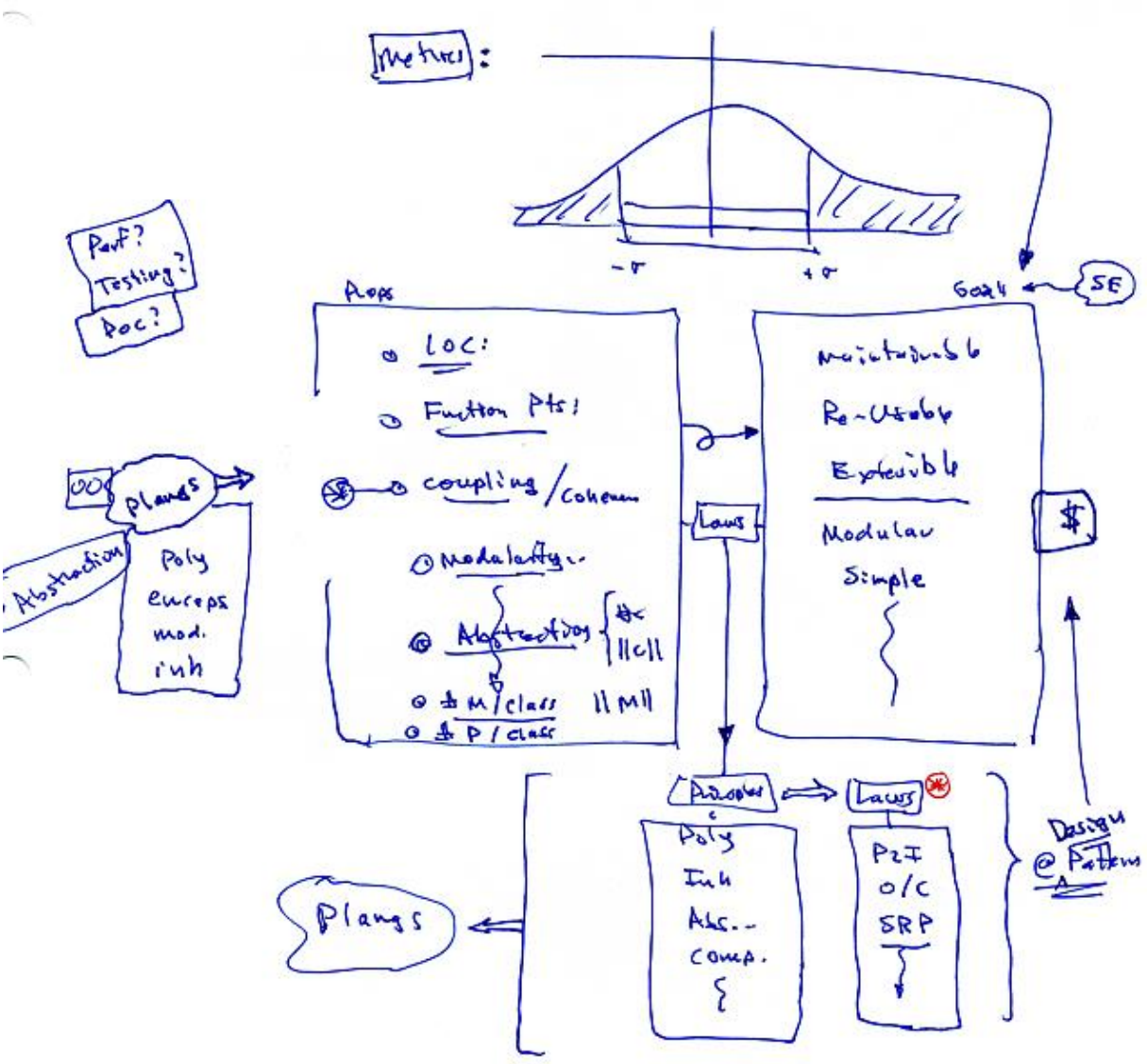
- Laws and Principles of SE & Design

Frameworks, Patterns, and Libraries







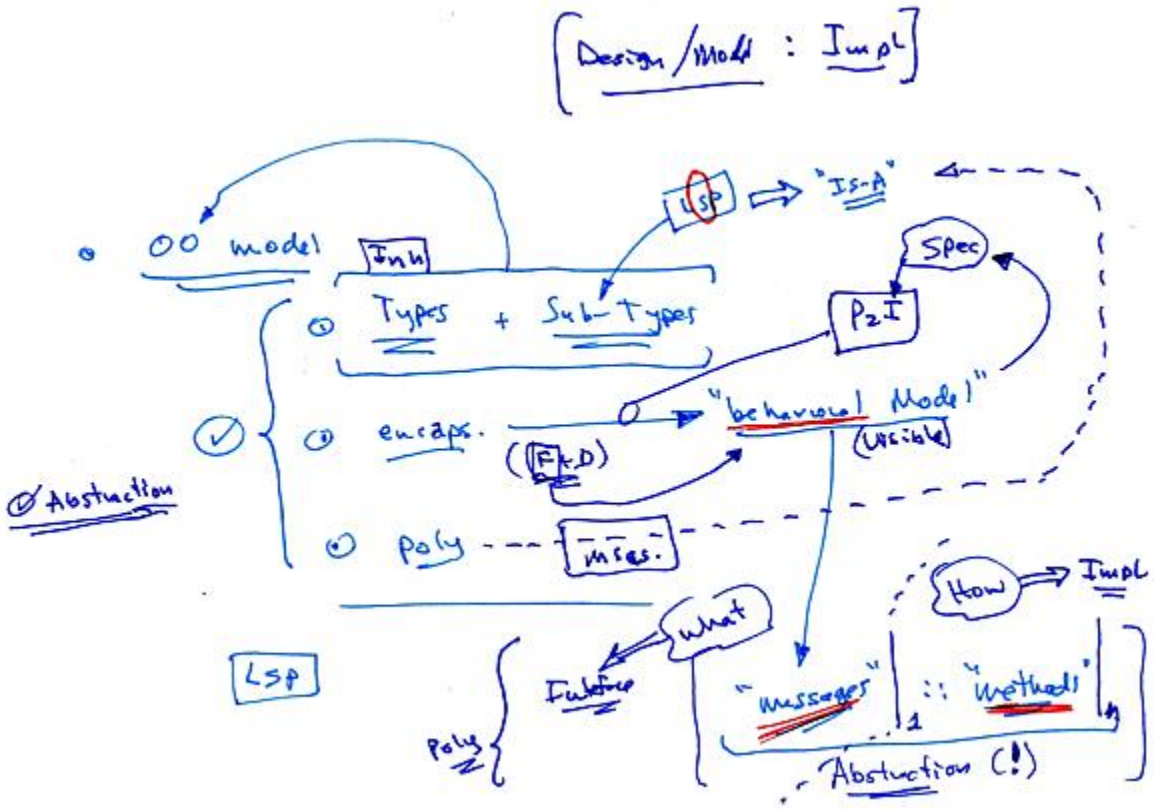


- Software Laws → Principles → Metrics

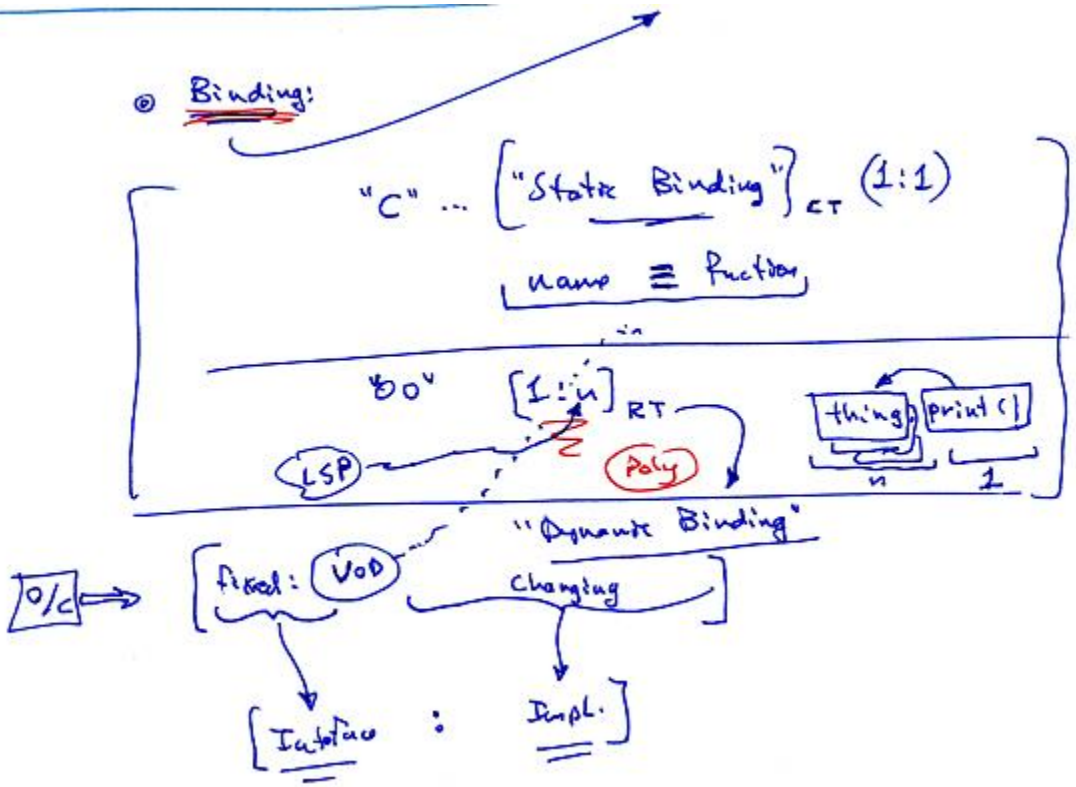
Day 2: OO programming Model and laws of Software & Design.

Day 2:

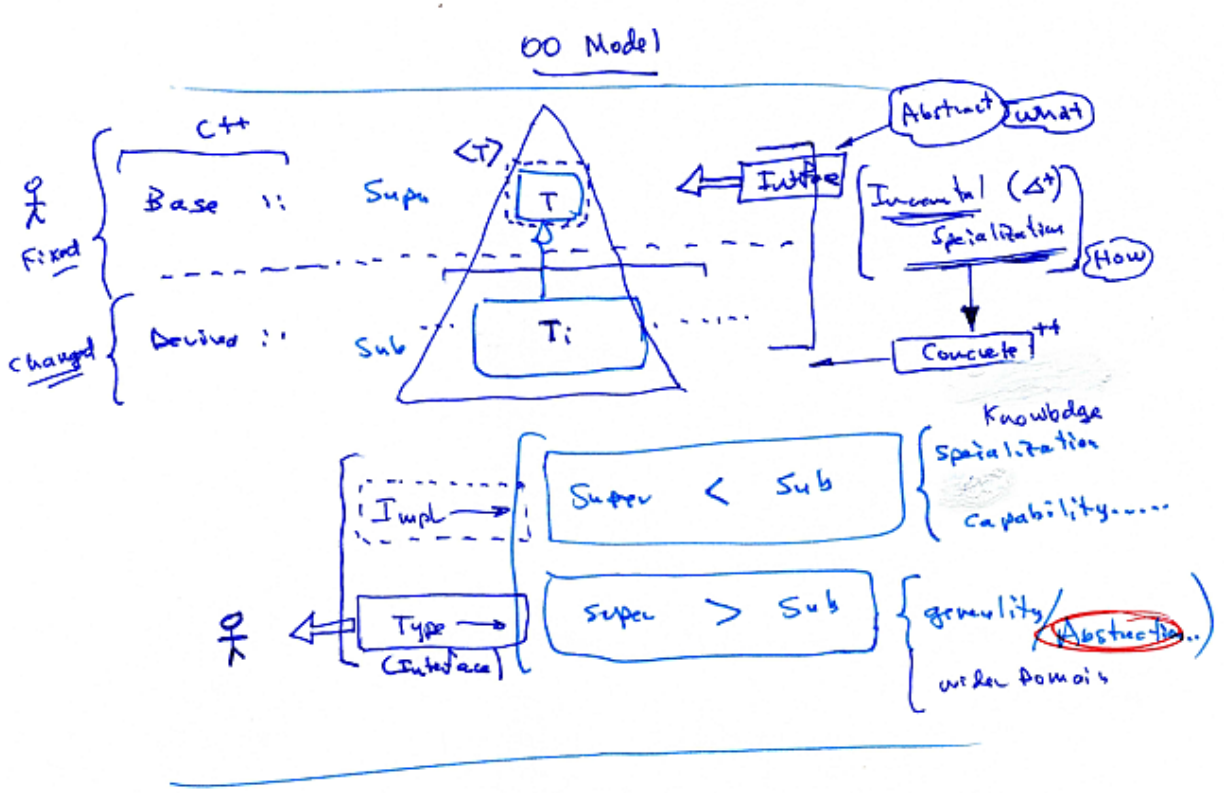
- Quiz & Review
  - Pattern languages ✓
- Refactoring ✓
- Programming Language Abstractions
  - OO; terminology, concepts
  - Types, Sub-type Abstraction
  - Polymorphism
- Principles ⇨ patterns
  - Variation Oriented Design (*VOD*)
  - P2I & LSP
  - decoupling, encapsulation, abstraction
- Patterns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



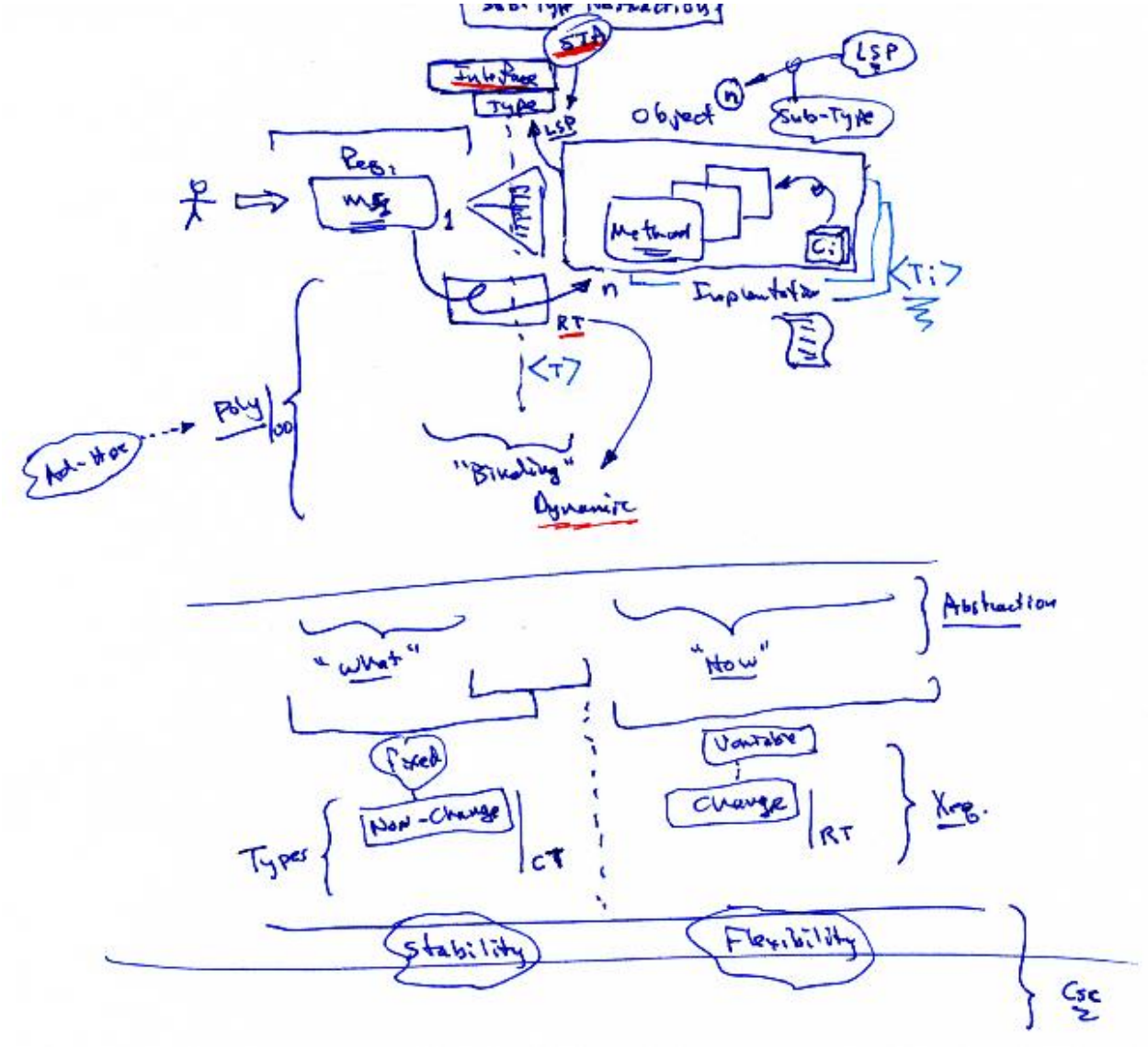
- Definitions and features of OO software



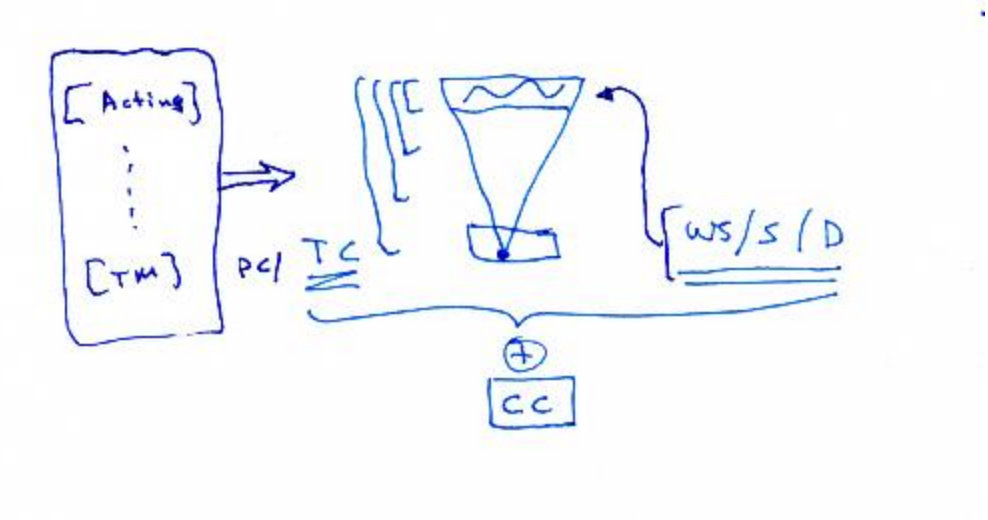
- Dynamic Binding



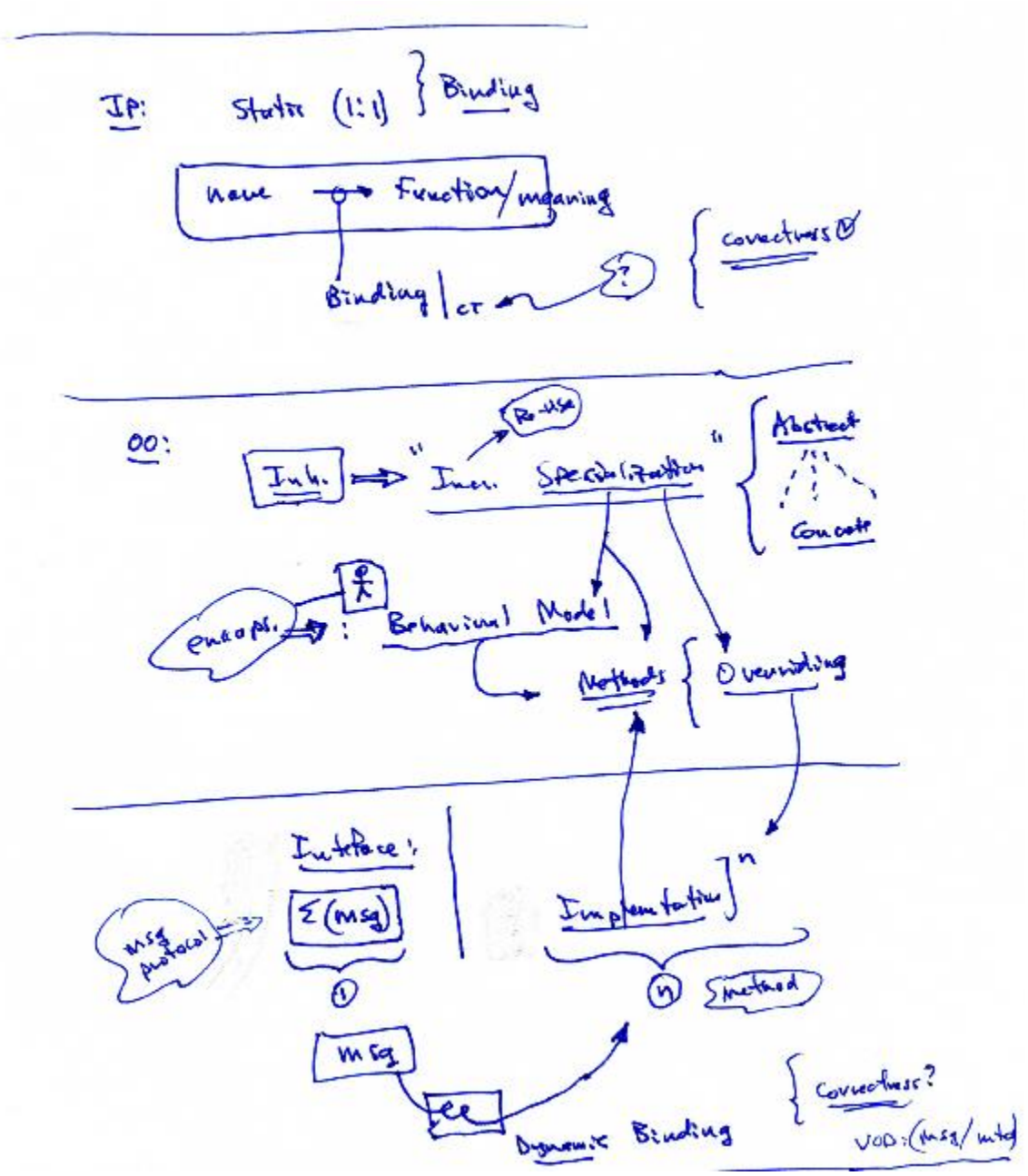
- Class Hierarchies



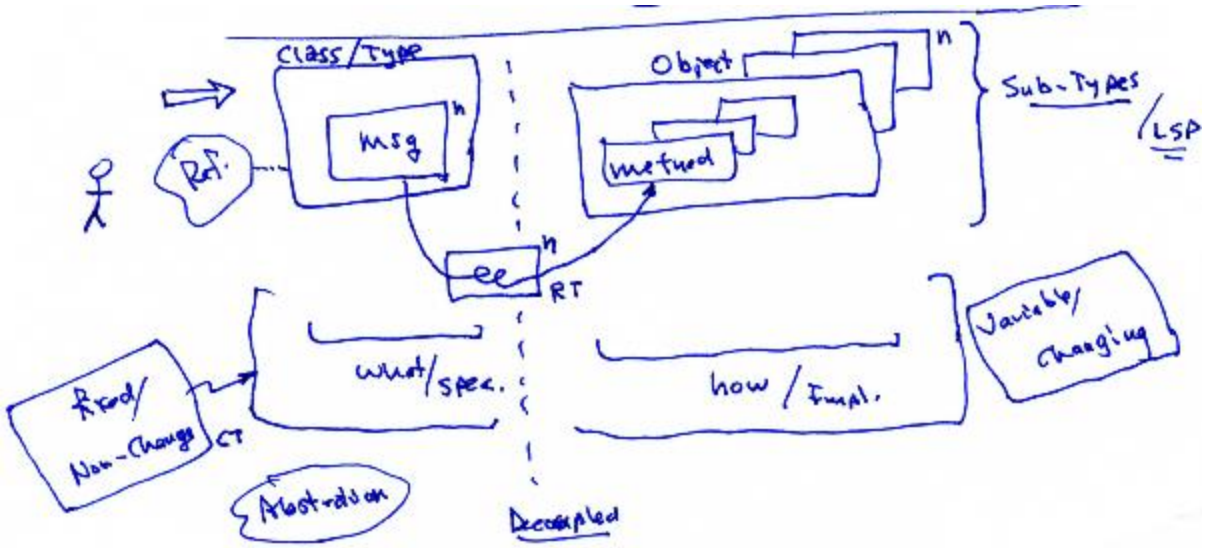
- Sub-type abstraction (STA)

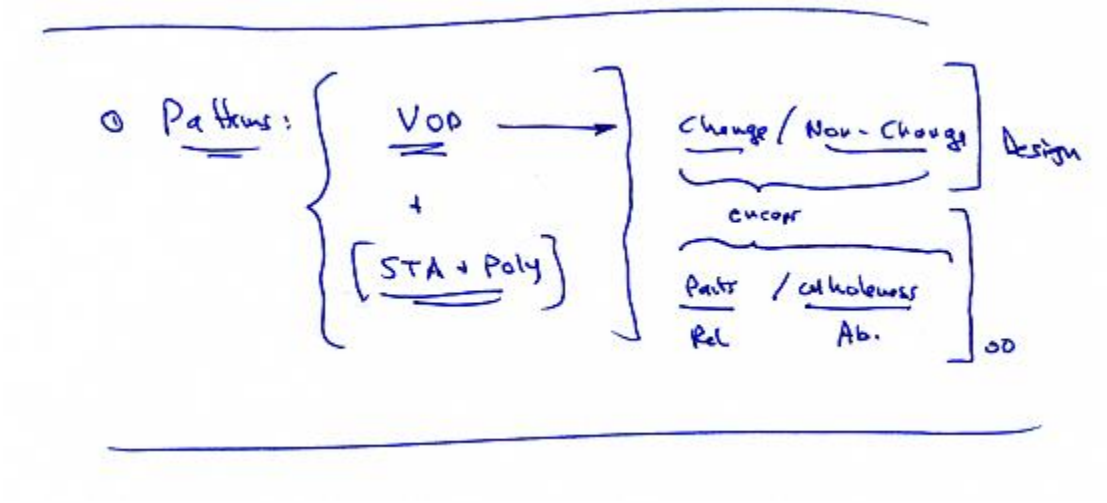


- SCI relationship to STA

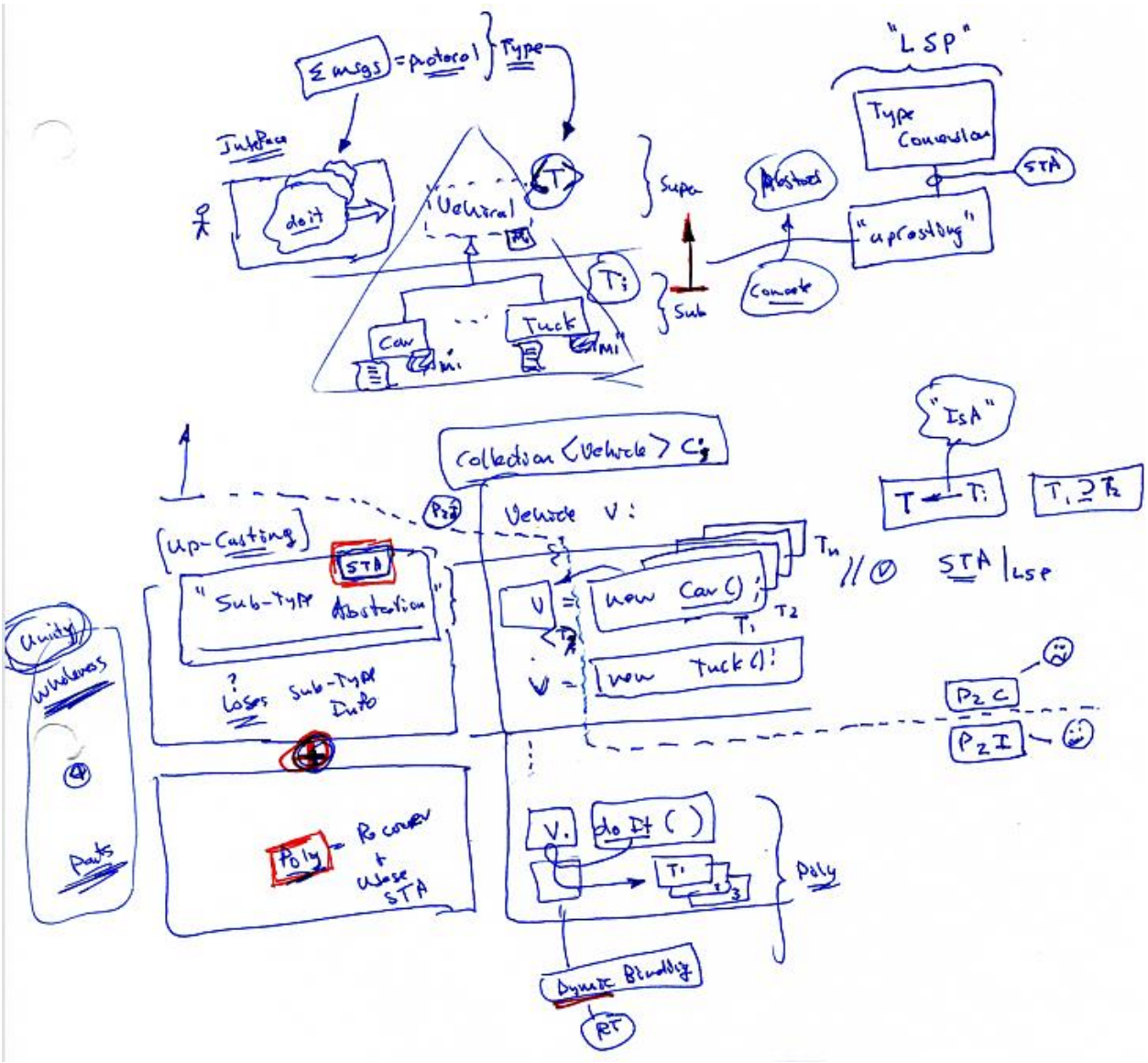


- Dynamic Binding

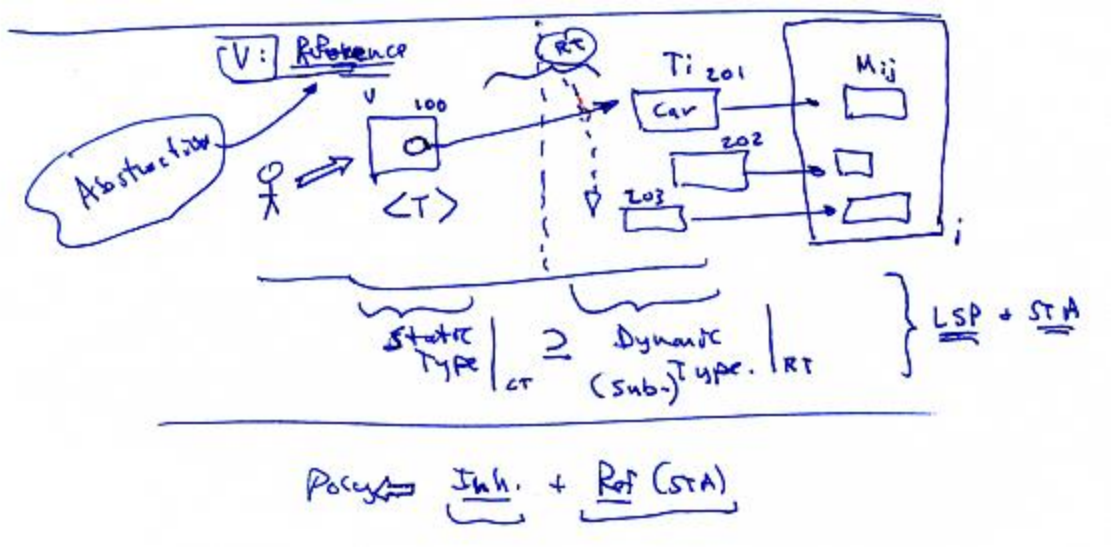




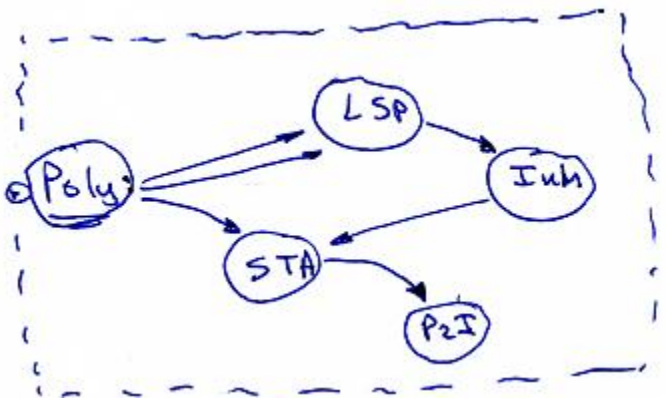
- The common basis of all patterns
- VOD = Change + non-change



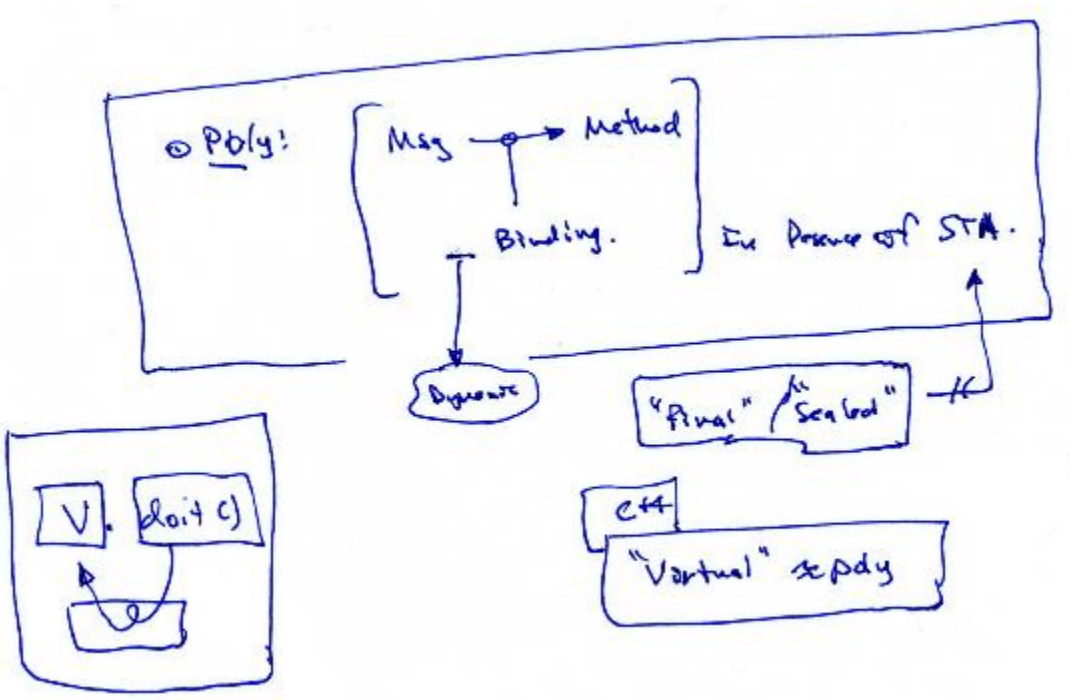
- STA + Poly ⇨ Wholeness + Parts
- Example of STA
- STA = *upcast*



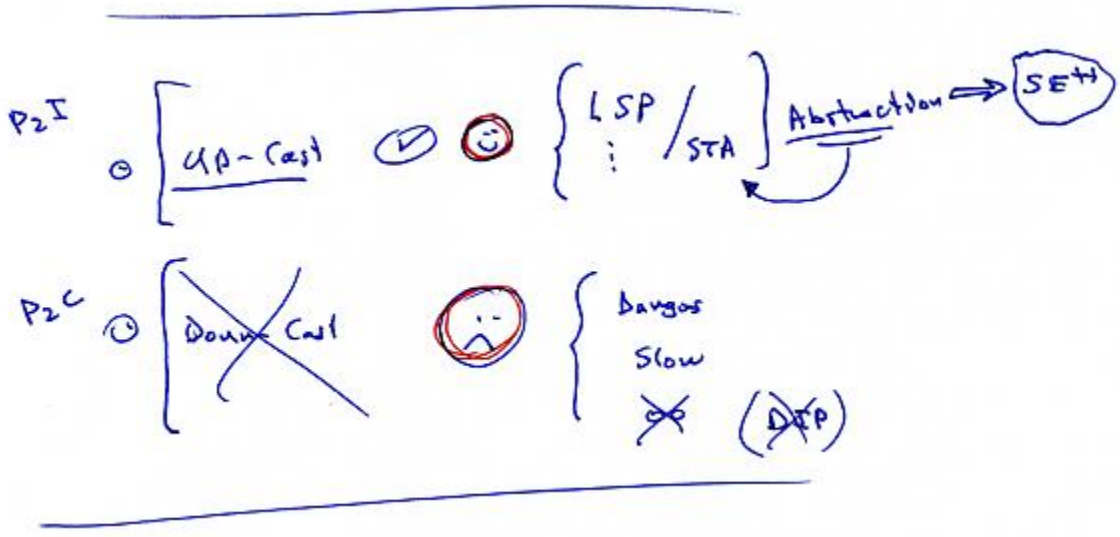
- References are an abstraction over values
- Poly & STA always/only occur together



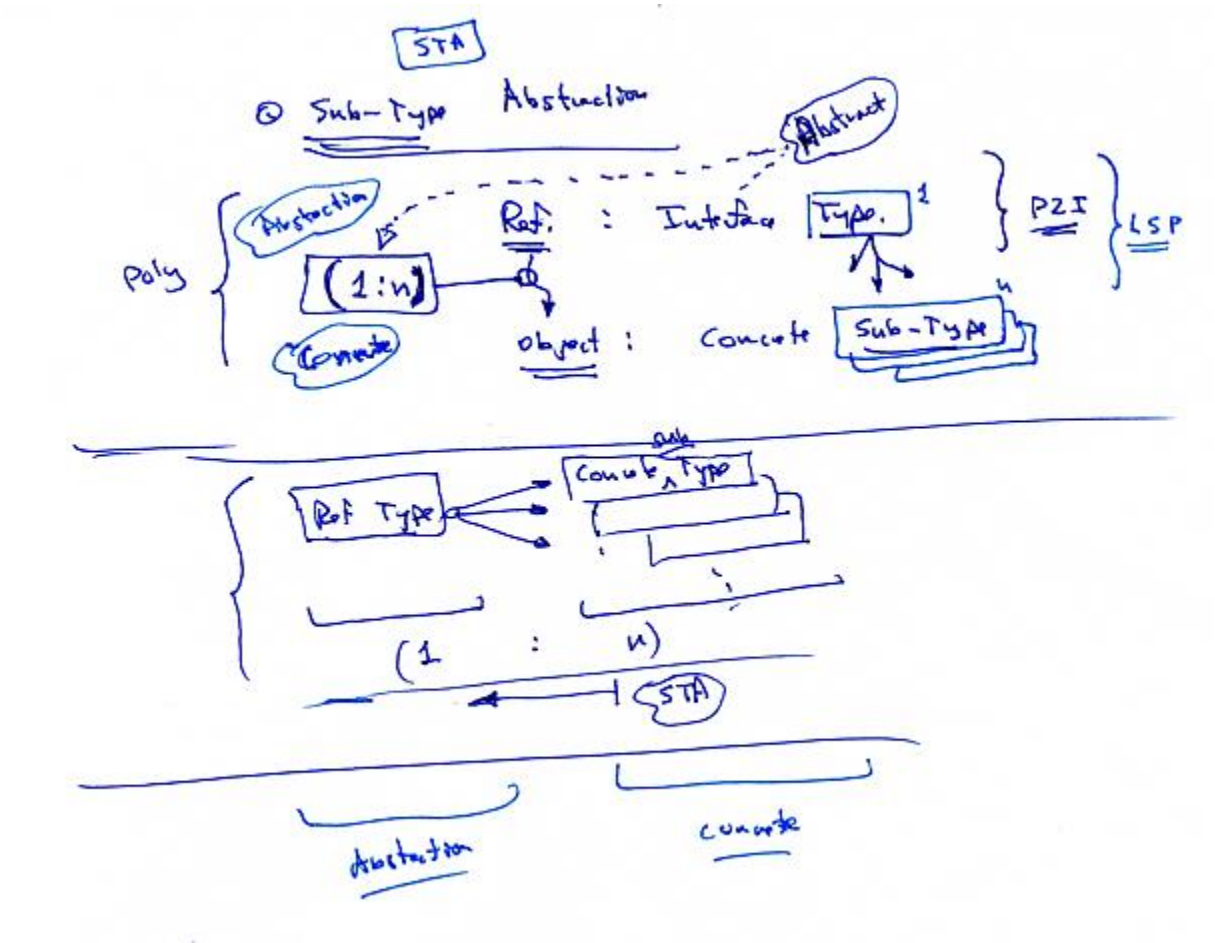
- Cluster of related concepts



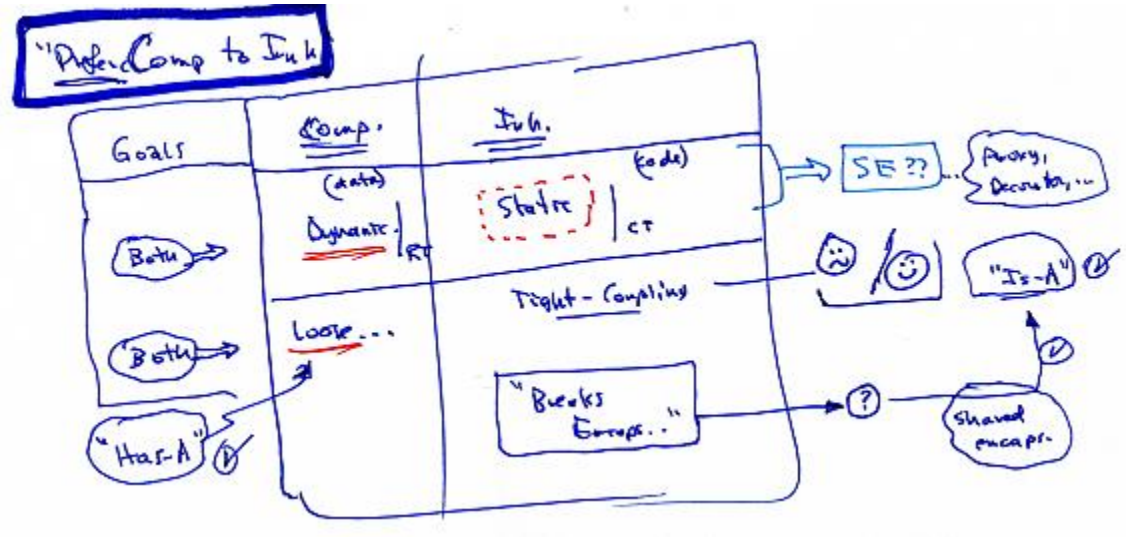
- Technical definition of polymorphism



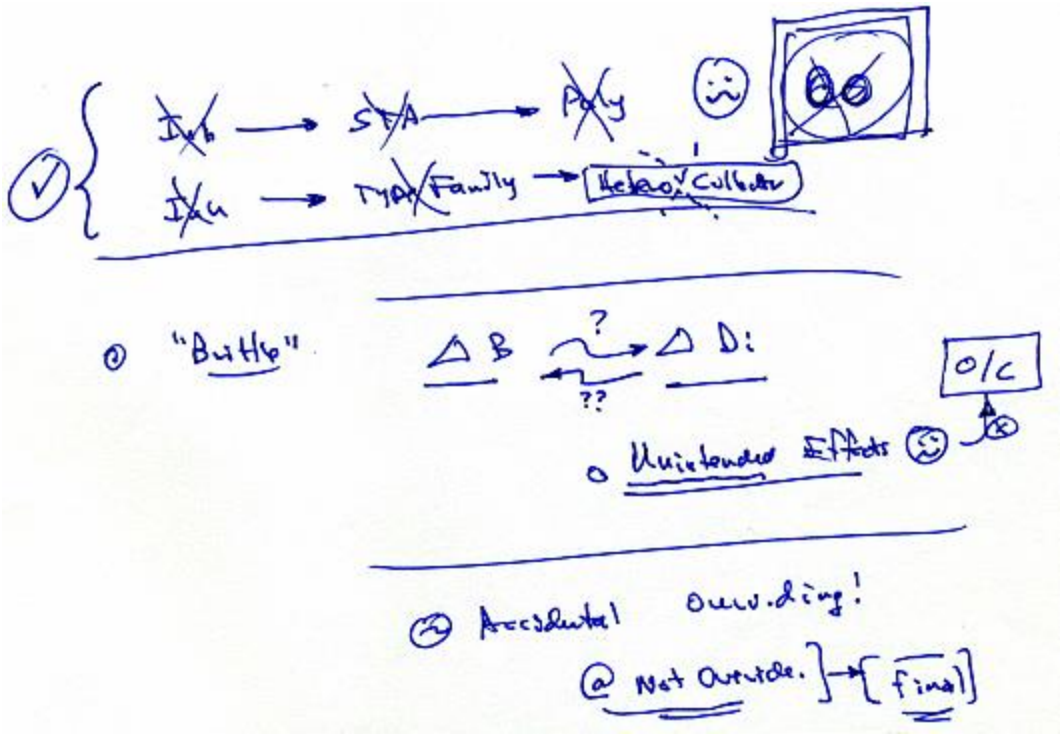
- LSP + STA = up-casting = Good,
  down-casting … not good



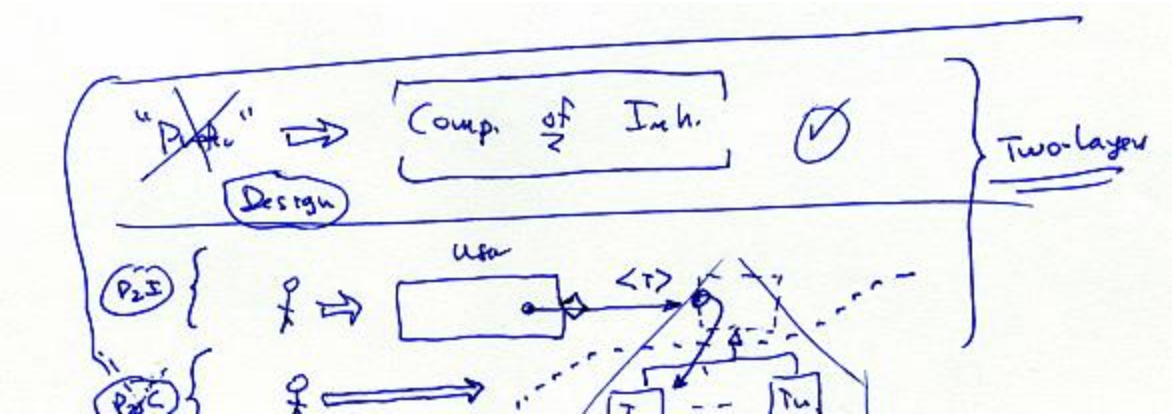
- P2I & STA



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



- Without inheritance; no LSP → no STA → no Poly ⇨ no OO!
- There are issues with inheritance – *Brittle* class relationships



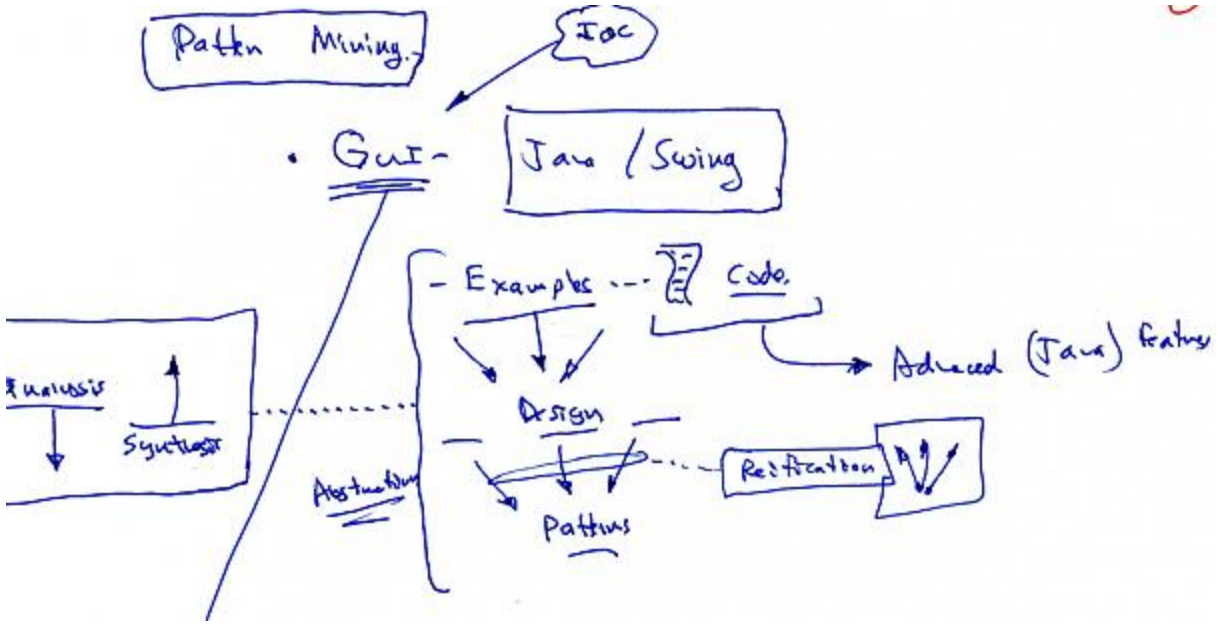
- Both composition & Inheritance (*Has-A, Is-A)* are important design concepts, and both should be suupported and used in design.

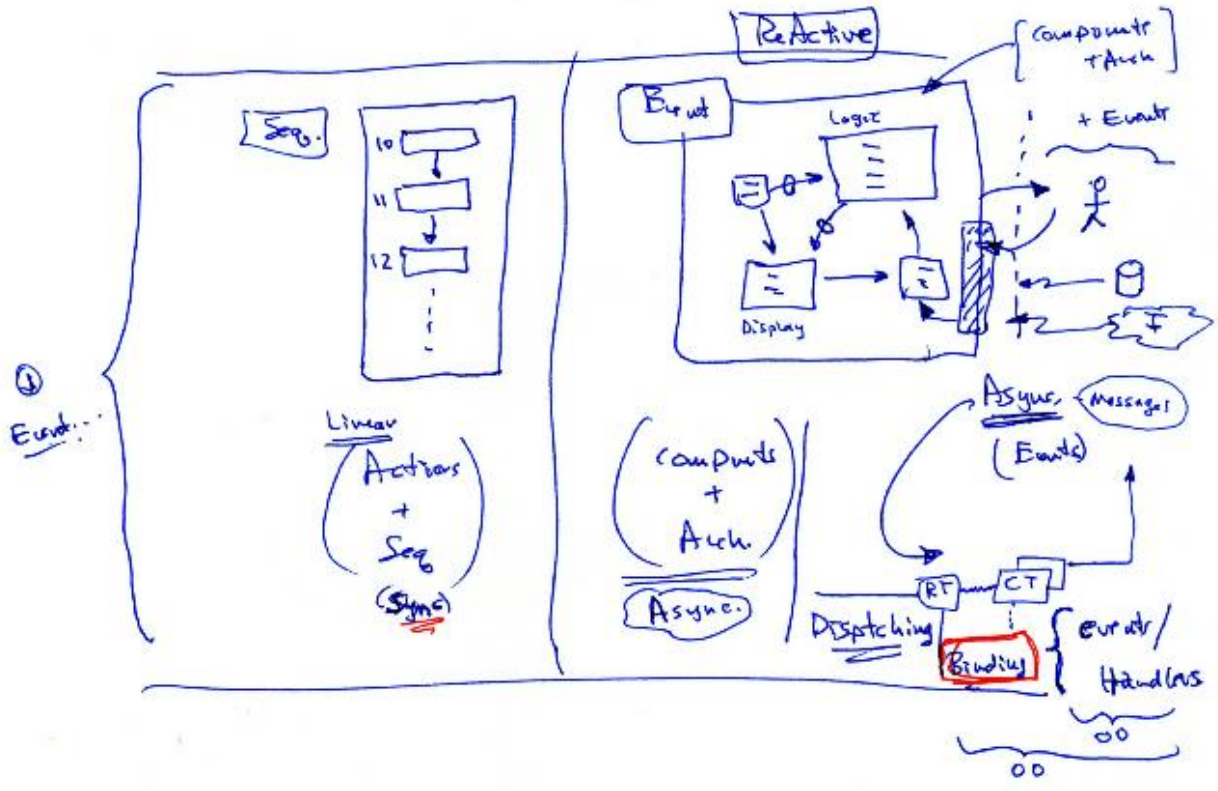
Day 3: Pattern Mining & Swing, Observer Pattern

Daily Topics

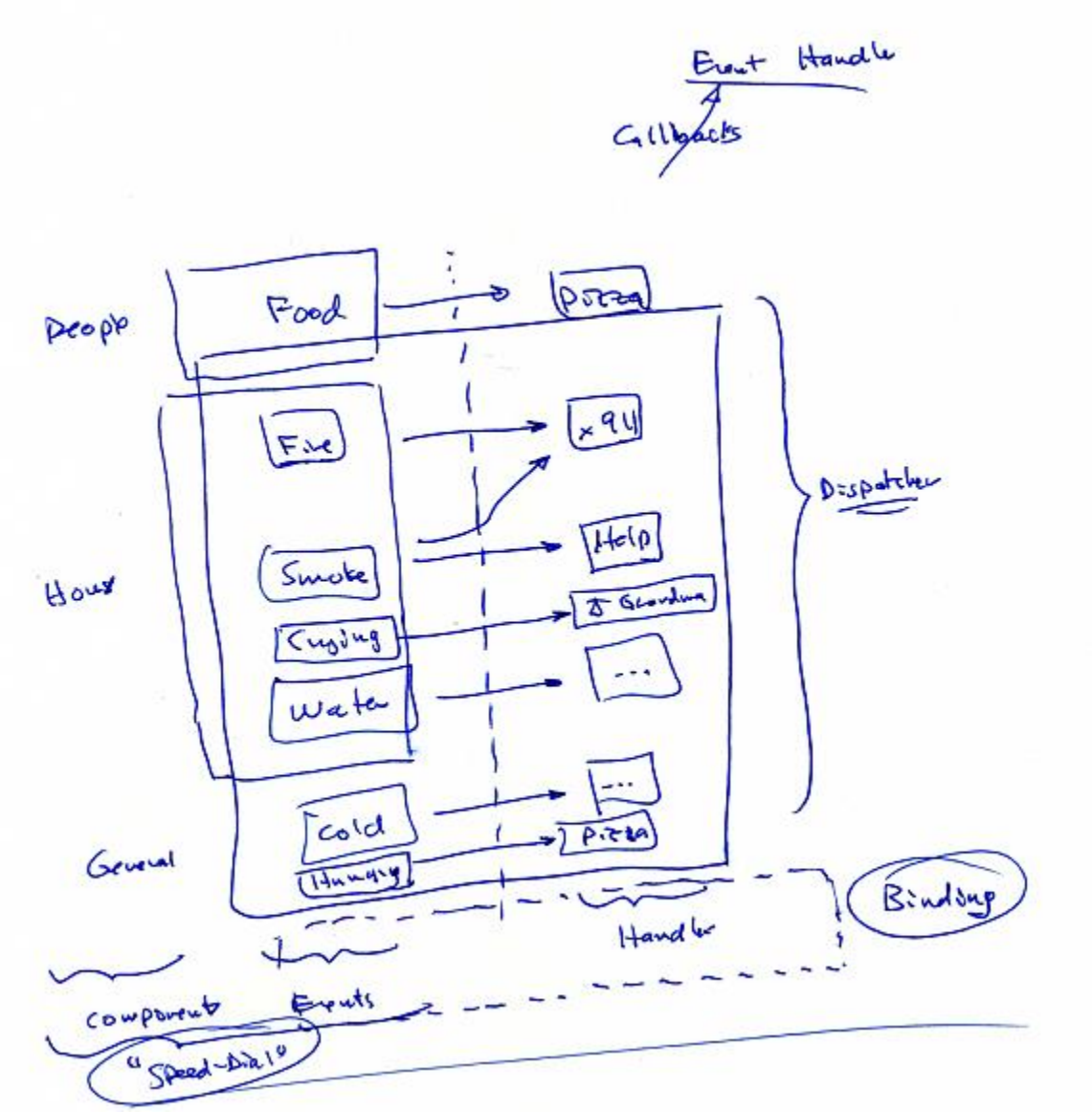
Day 3: Pattern Mining & the Observer Pattern

- 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 ⇨ GOF
  - Standard pattern template
- Patterns in Swing:
  - MVC
  - command
  - observer
  - Composite
  - Strategy, Bridge, …

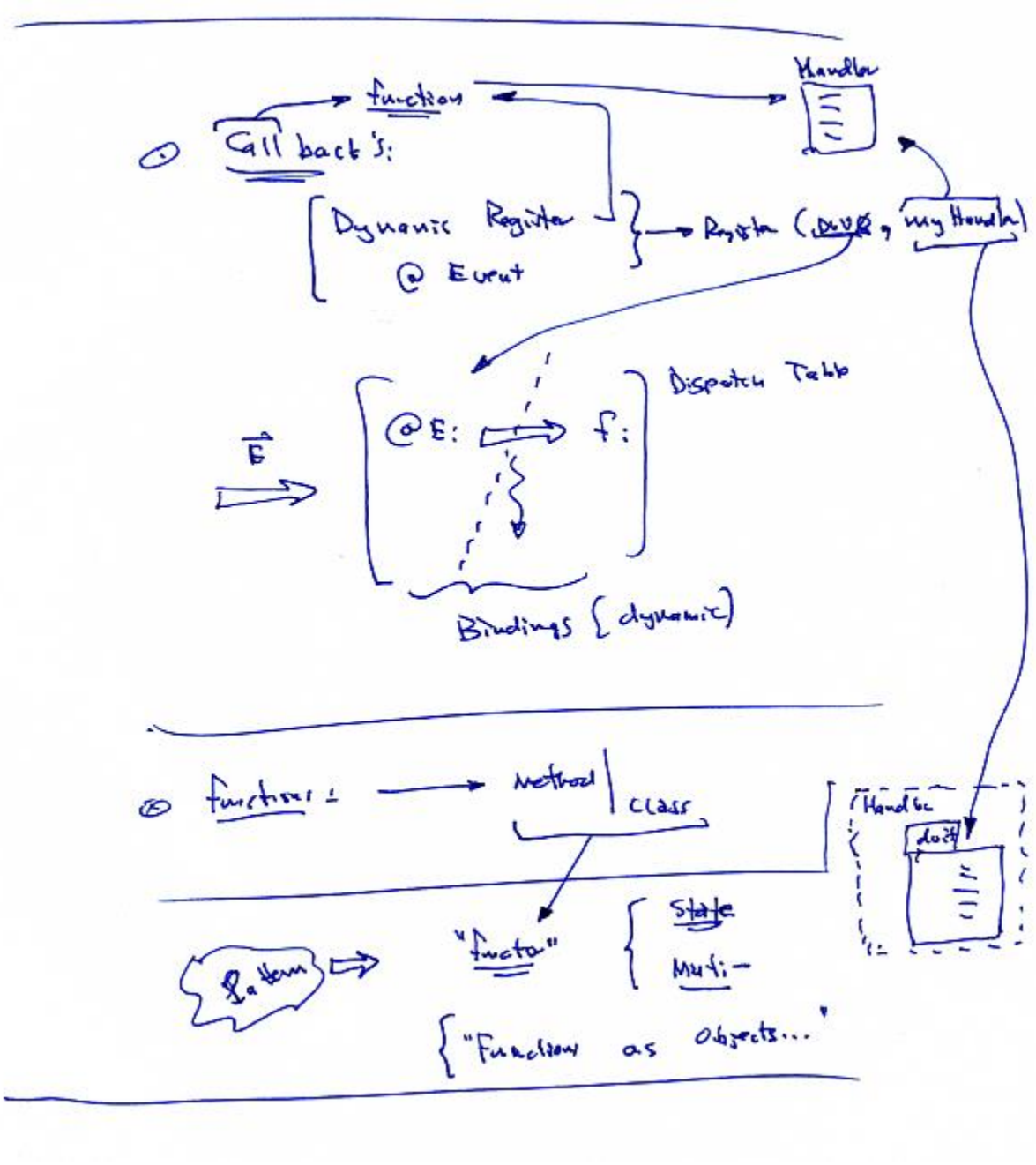




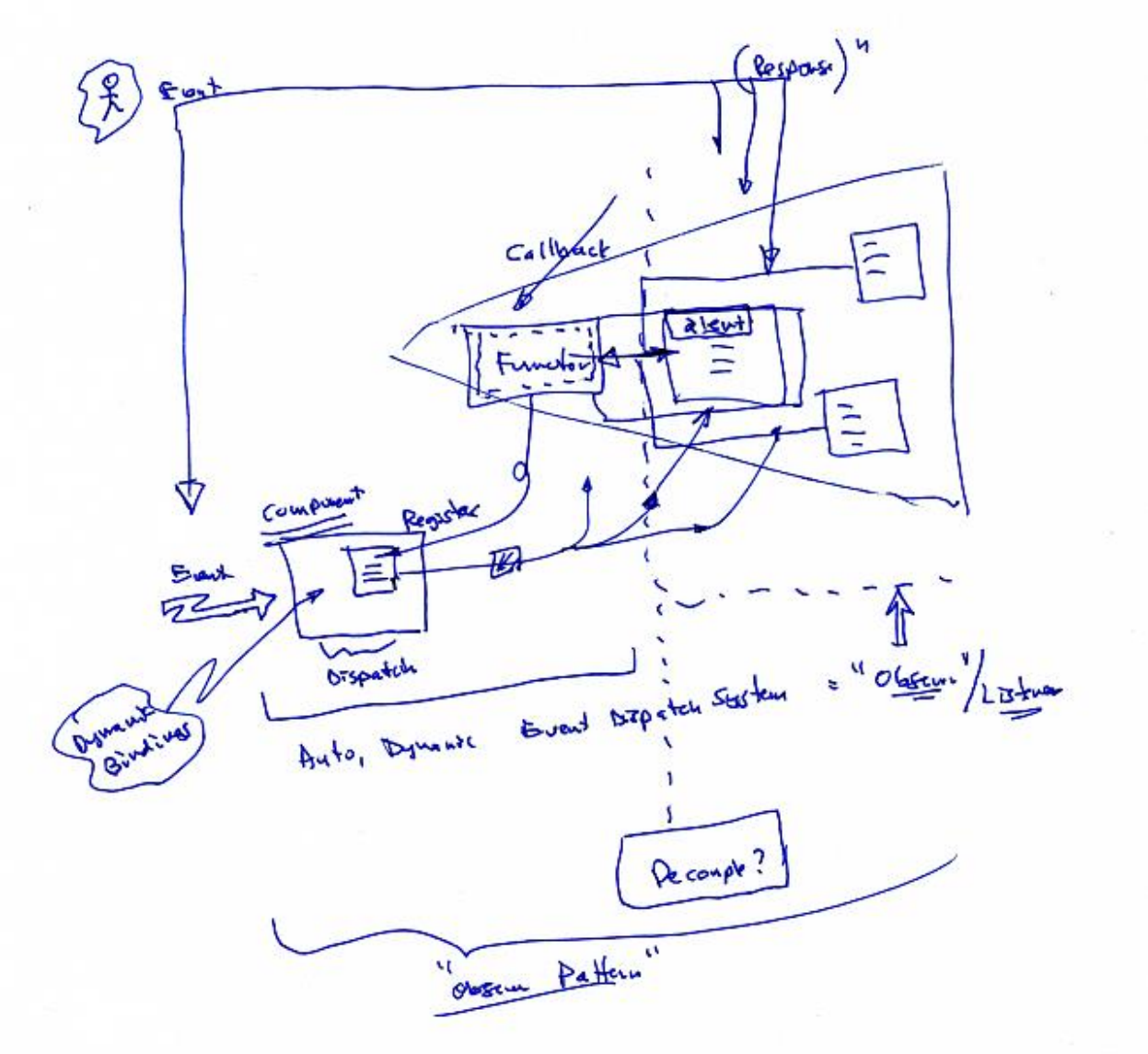
- Event driven programming model



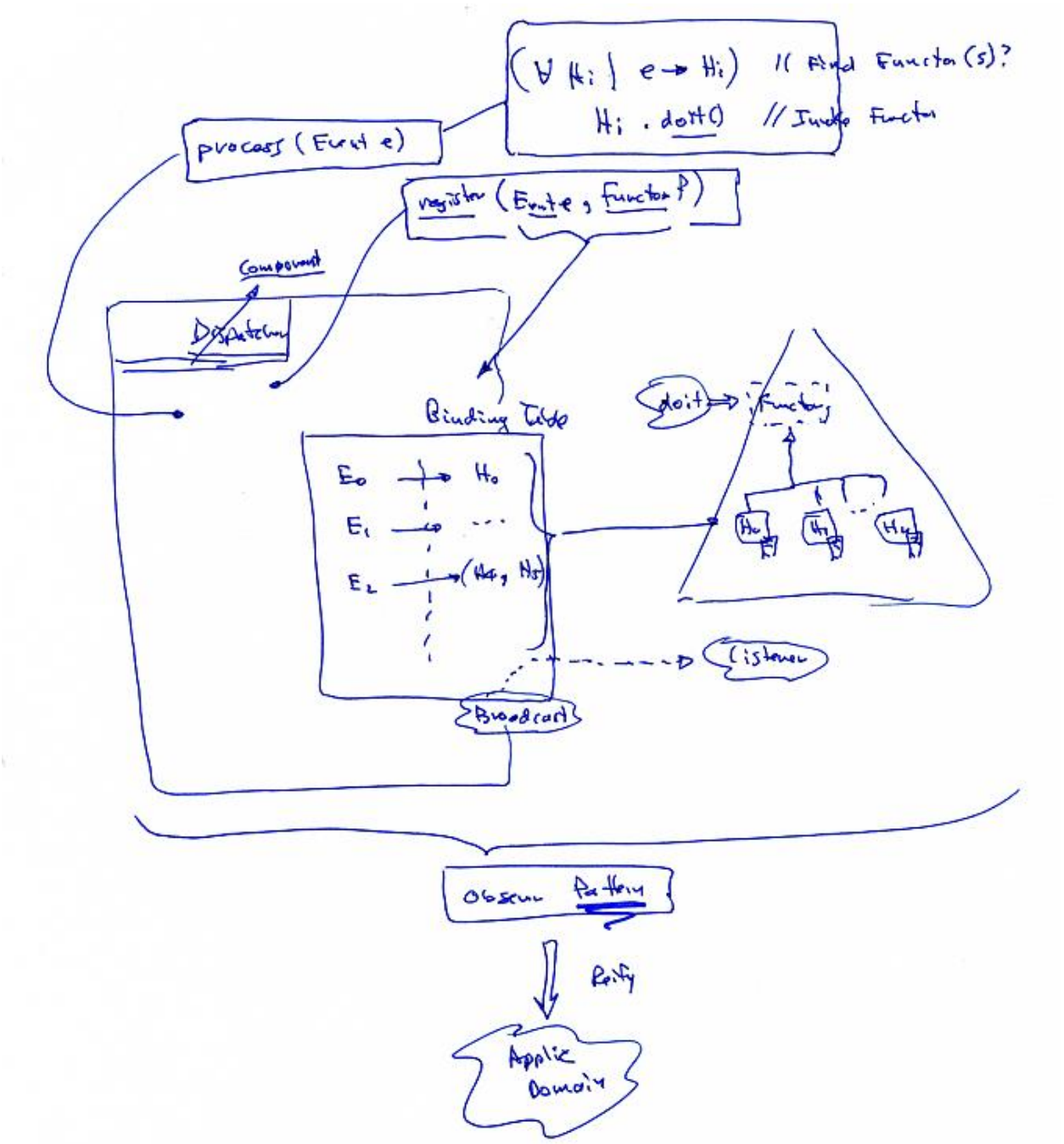
- Callbacks and event handlers



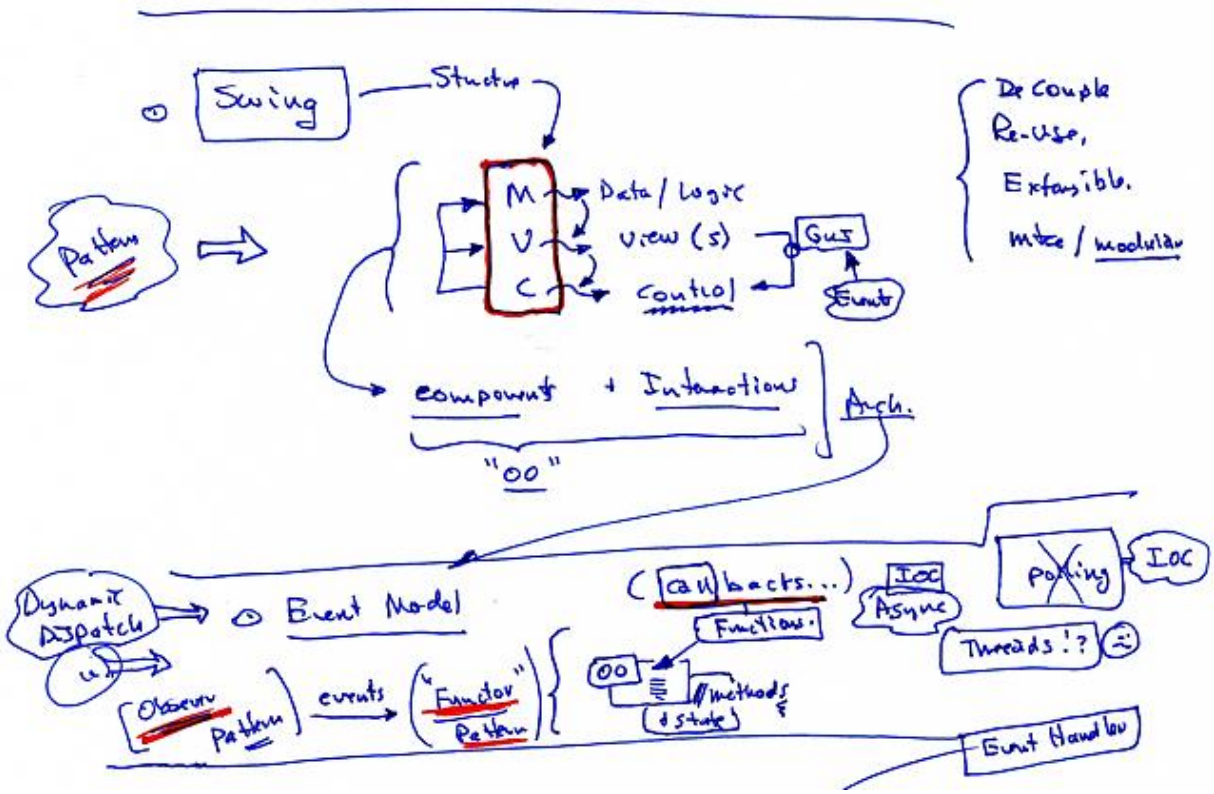
- Callbacks as event handlers
- *Callbacks* = functions ⇨ *functors* in OO



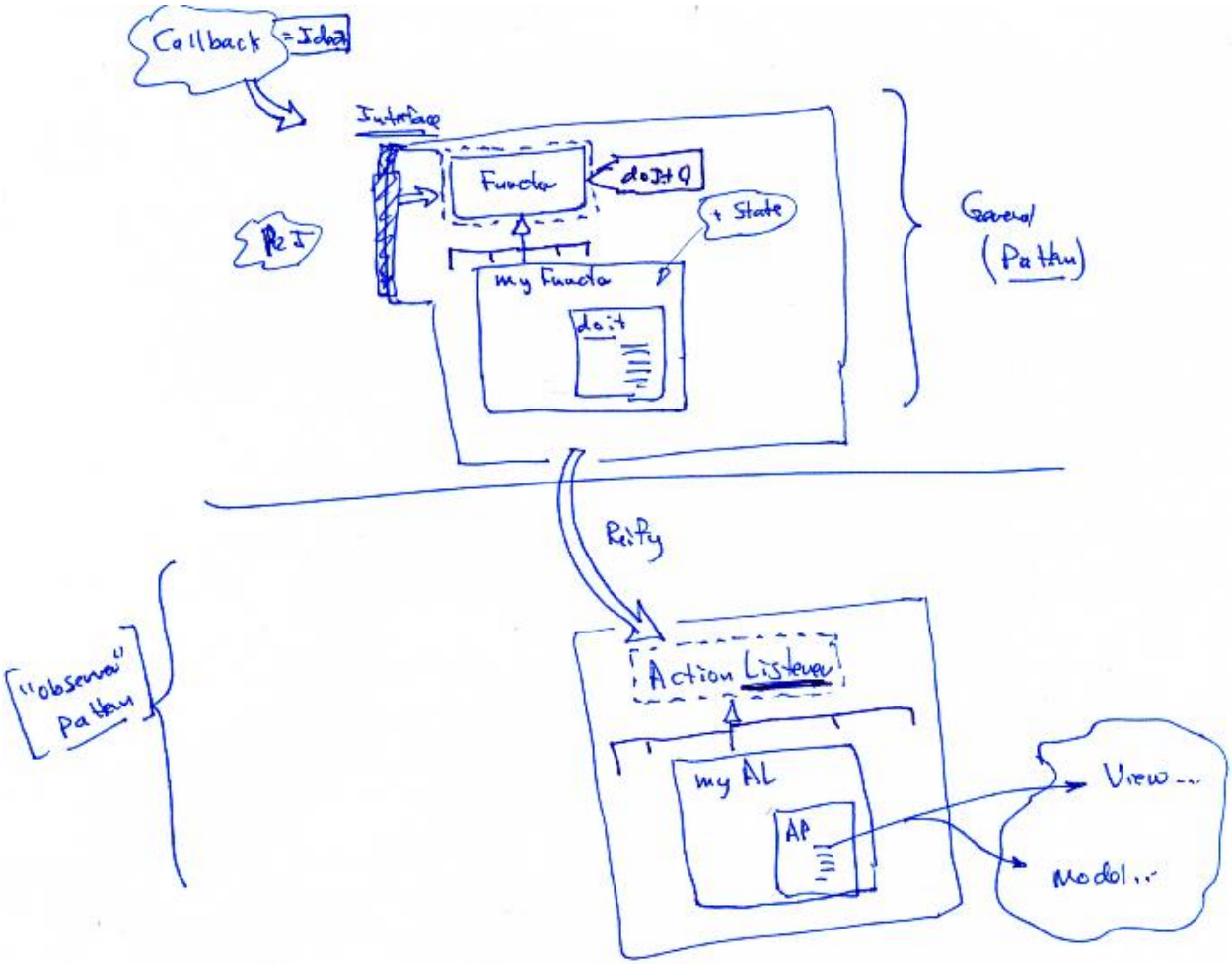
- Using a registry of functors (event handlers) for event dispatching



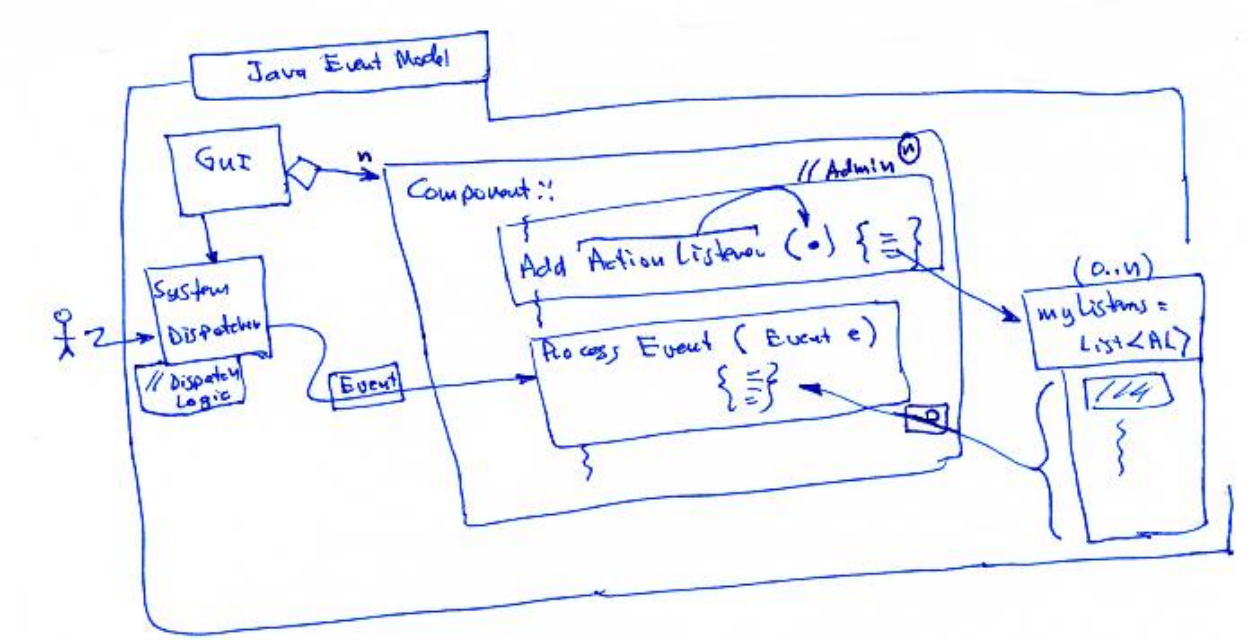
- ⇨ Observers (Listeners)
- + management and registration ⇨ *Observer Pattern*



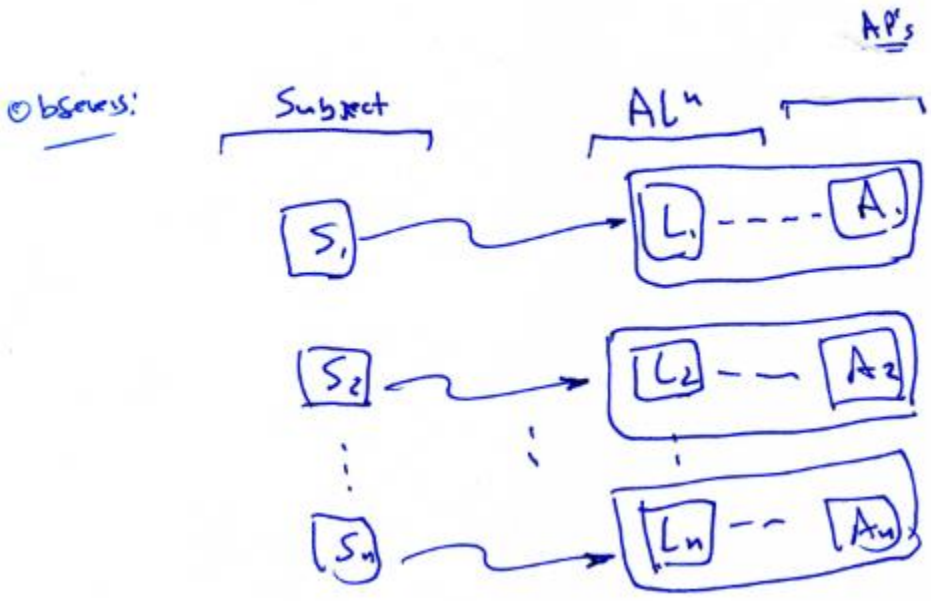
- Swing uses MVC and the Observer pattern



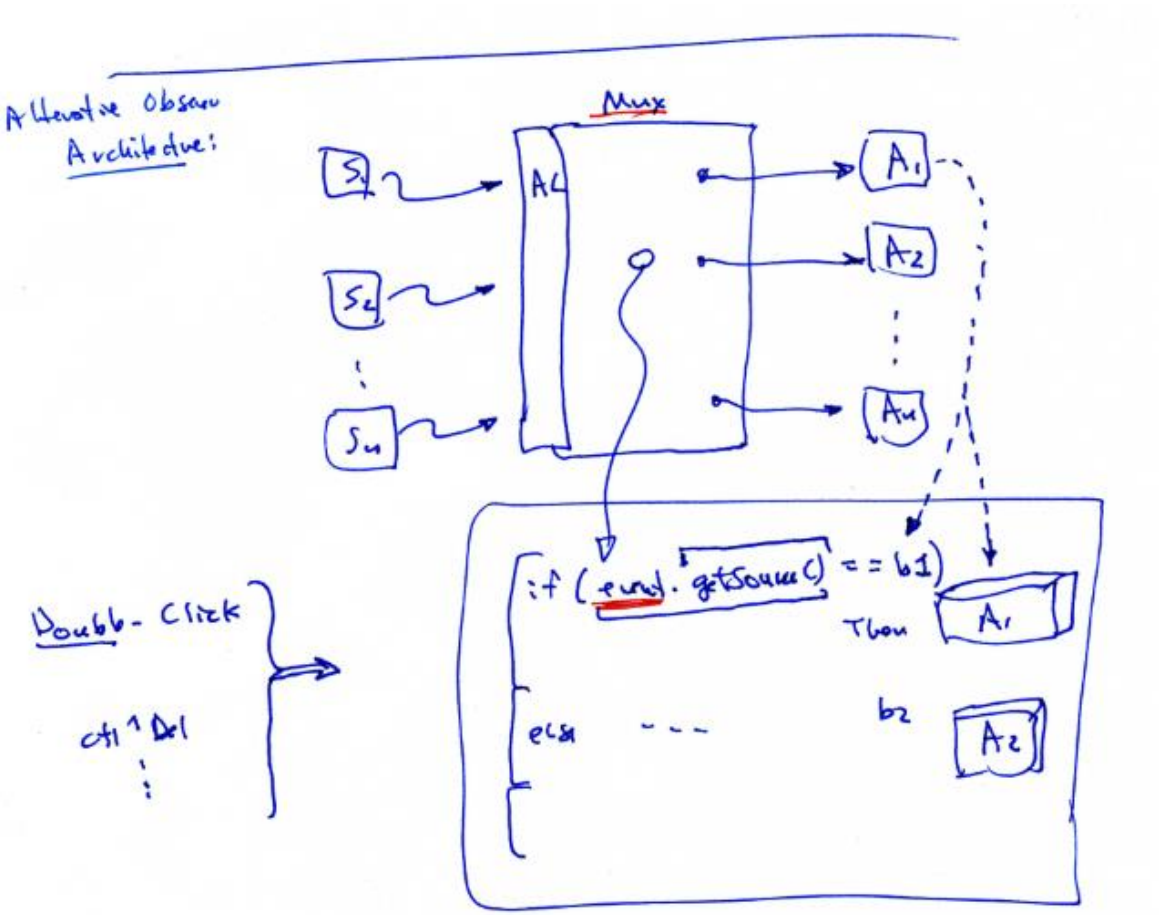
- Reify the General Observer Pattern → Swing ActionListeners



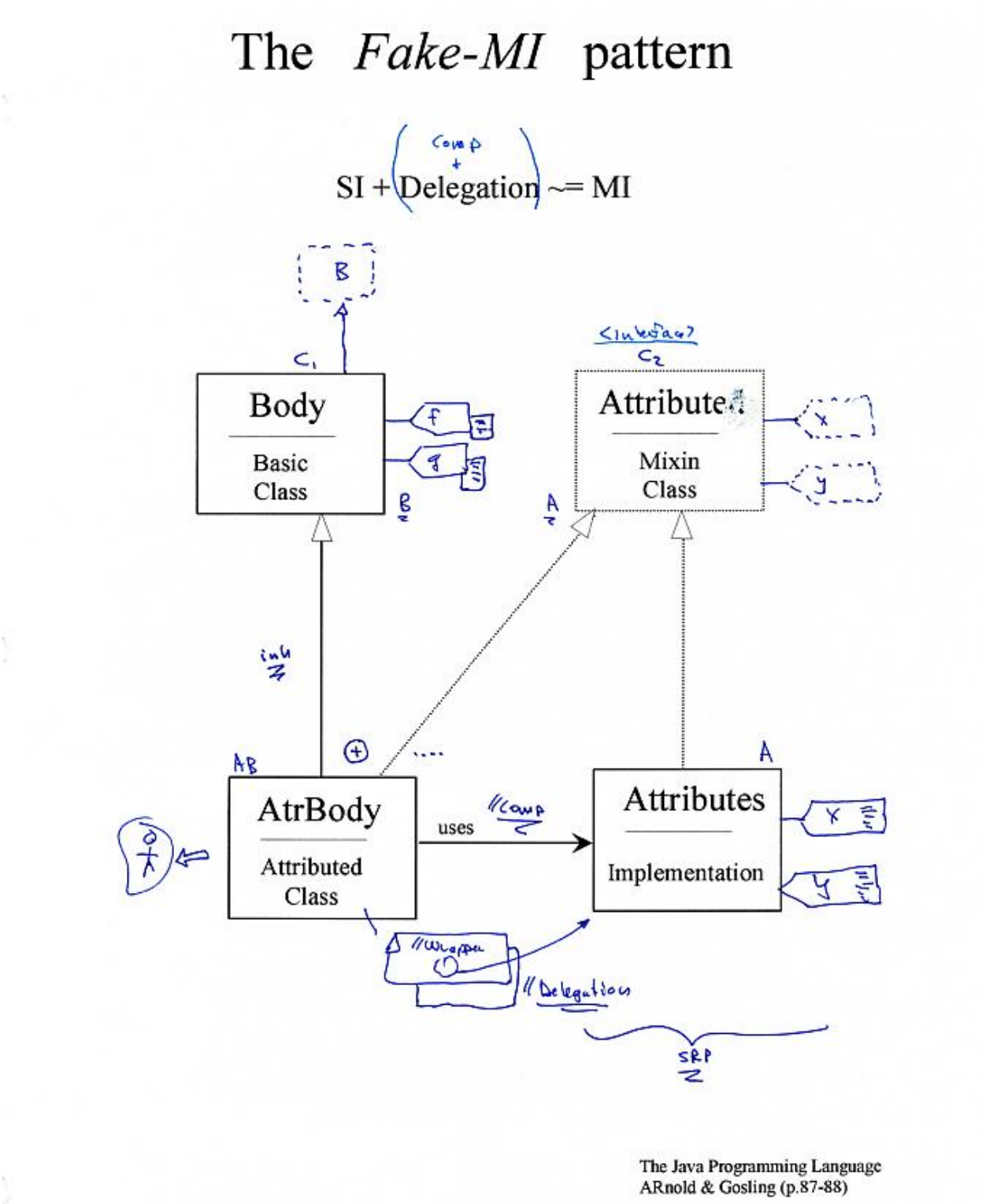
- Java Swing event model



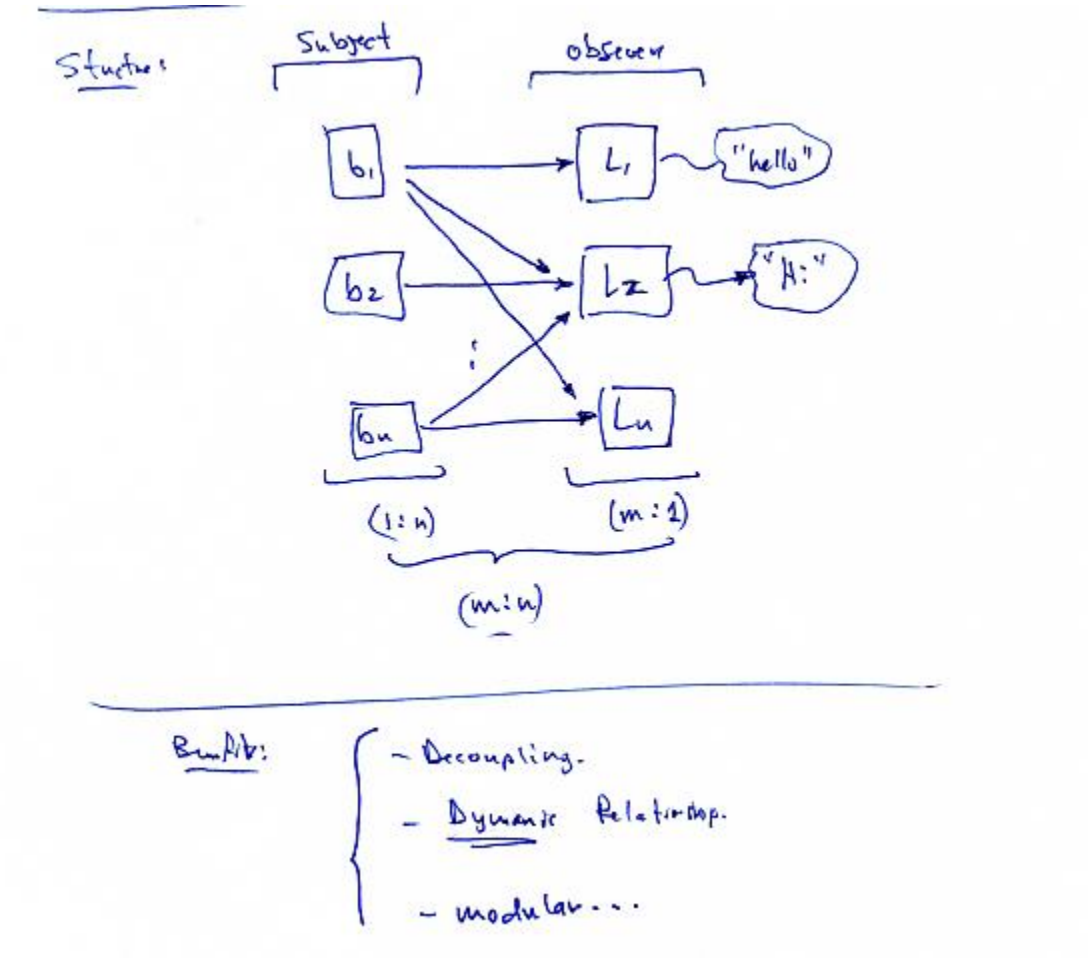
- Individual AL's @ event (subject) = de-multiplexed (like our class & GOF)



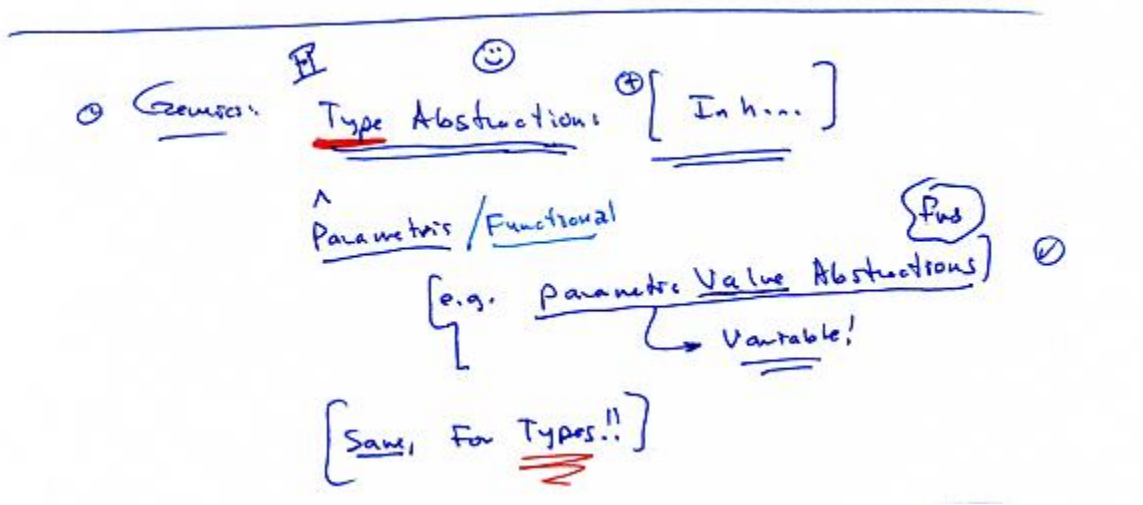
- Multiplexed Events (Event Manager), like the lab software



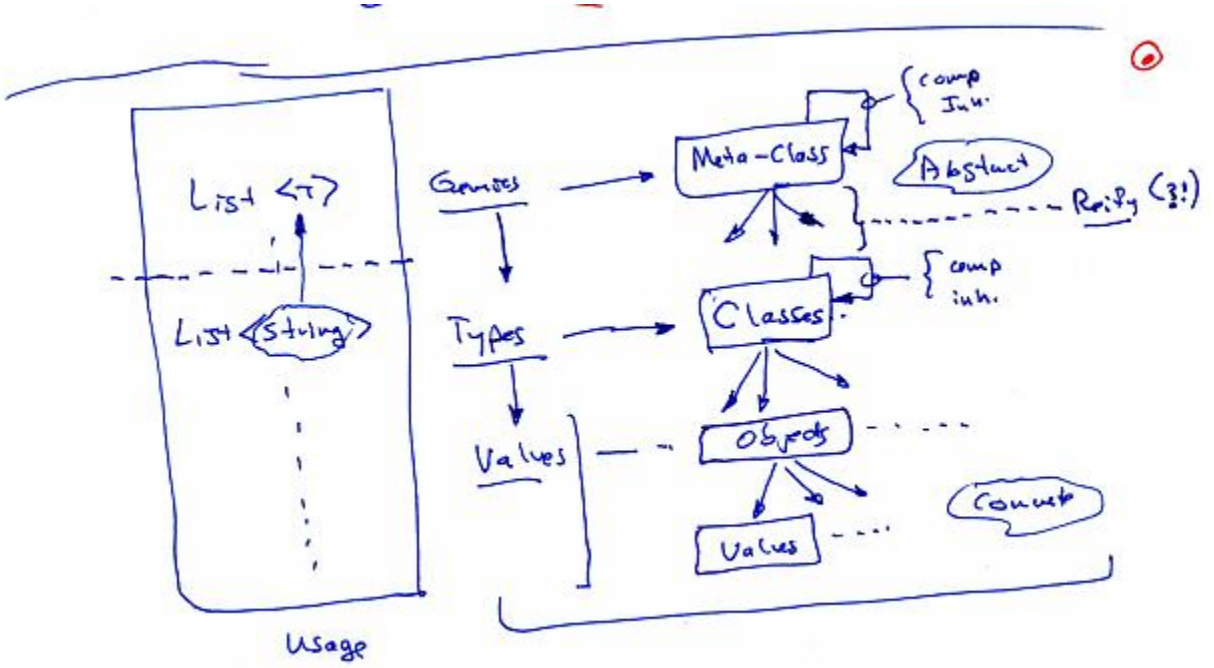
- In the Java library, the Observer pattern implementation has some issues
- The Subject is not an interface, so cannot be easily combined with any other classes for inheritance since Java only has Single-inheritance.
- Thus one has to use "Fake MI" using only SI



- (n:m) use of the Observer pattern



- 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

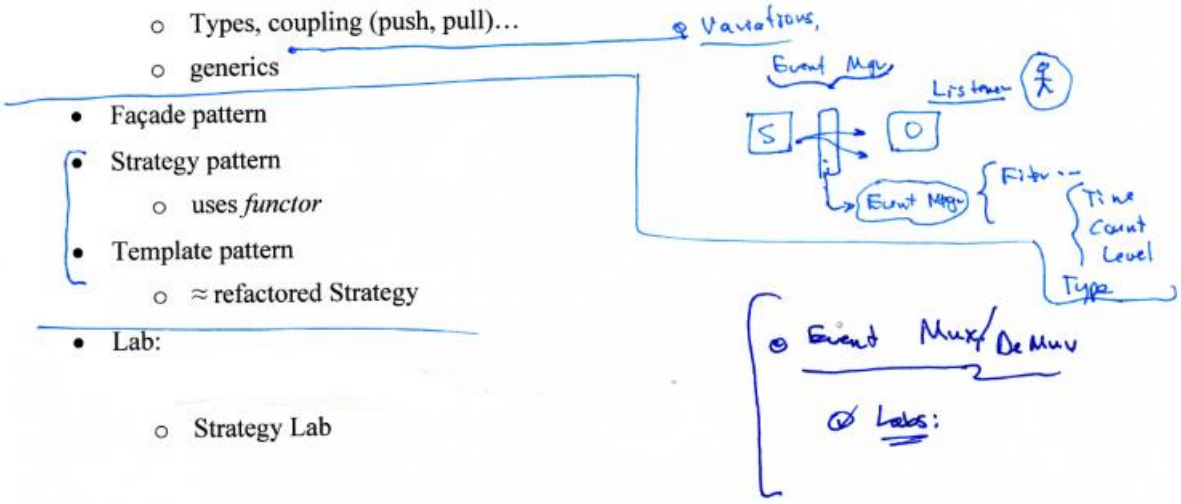
Day 4: Façade, Strategy, and Template Patterns.

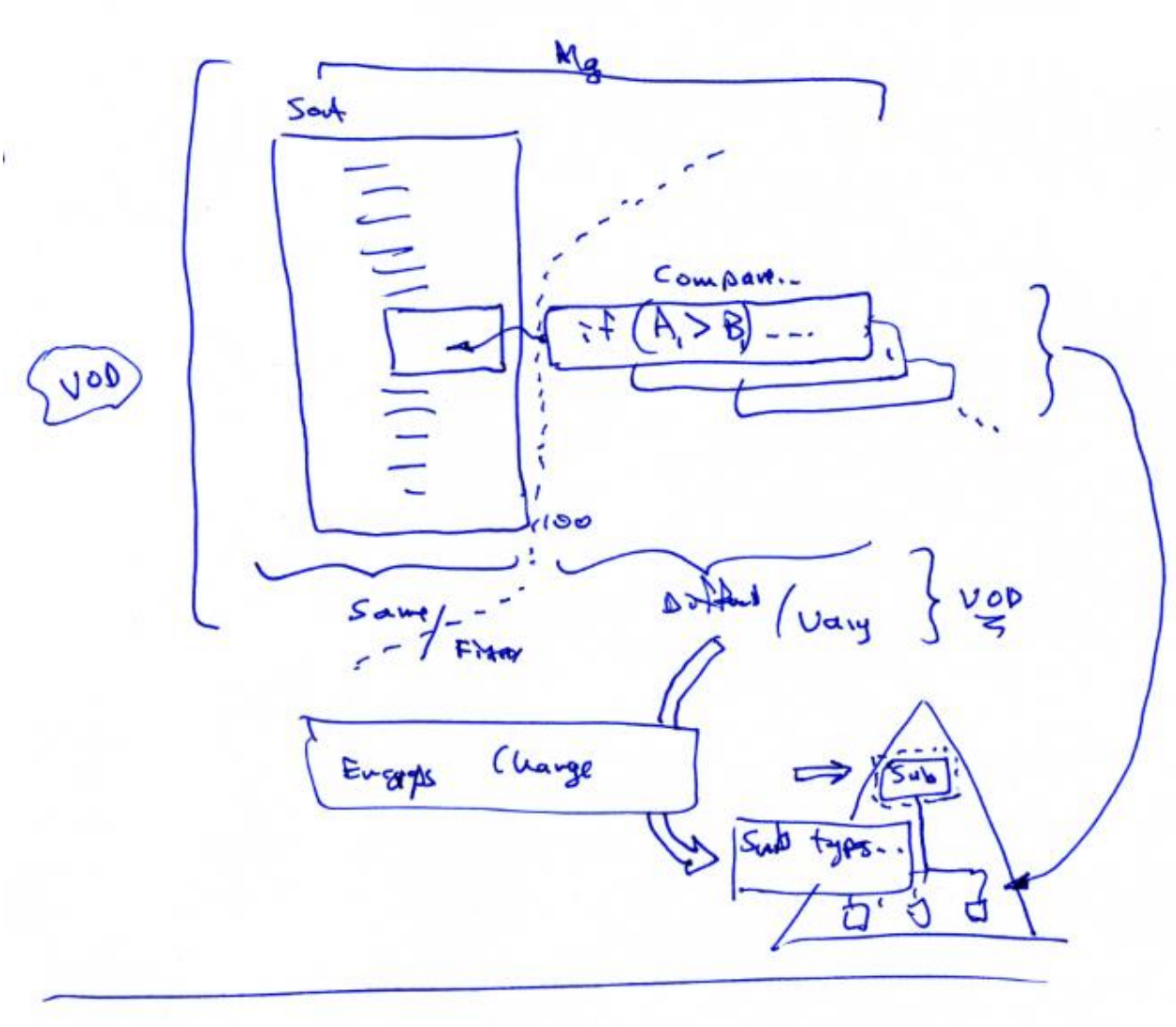
Advanced Software Development

Daily Topics

Day 4:

- Quiz & Review
- Observer
  - examples & issues
  - Java Implementation ∈ SI ⇨ MI
  - Types, coupling (push, pull)…
  - generics
- Façade pattern
- Strategy pattern
  - uses *functor*
- Template pattern
  - ≈ refactored Strategy
- Lab:
  - Strategy Lab

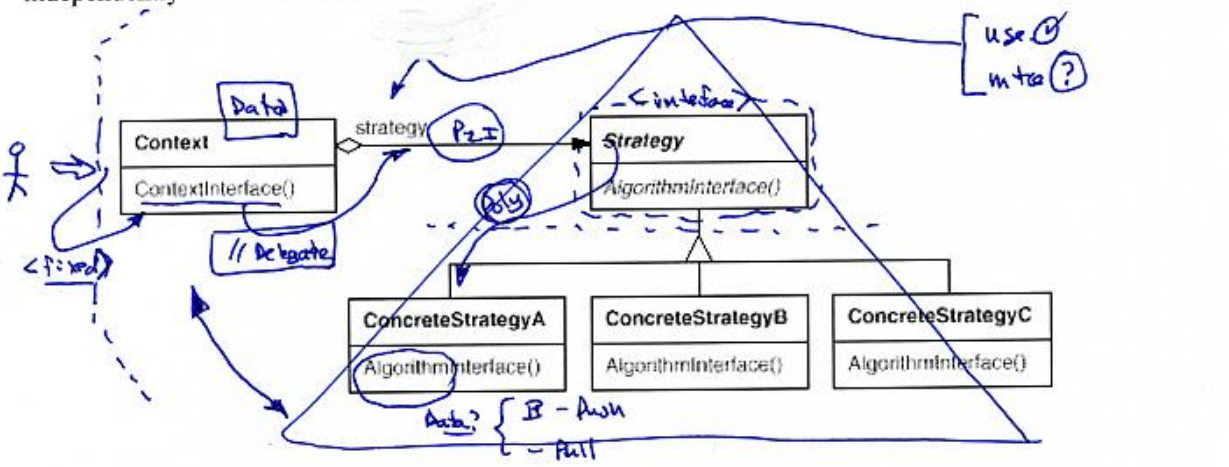




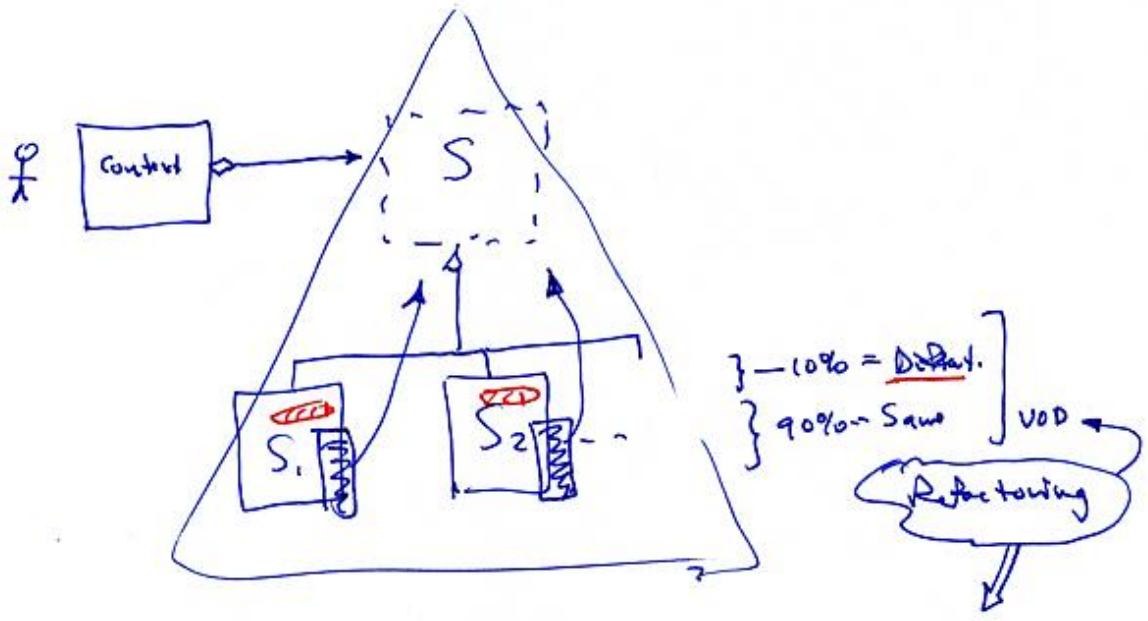
- The strategy pattern uses VOD to isolate and encapsulate changing parts of an algorithm.

Strategy Pattern

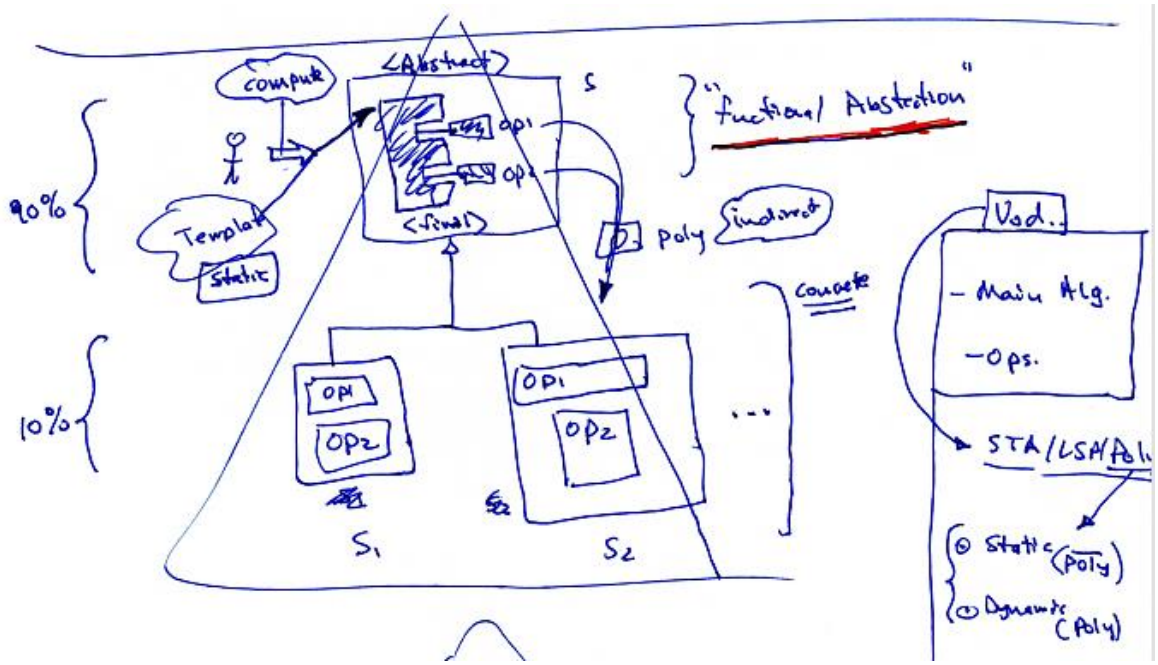
Intent:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Useful when you want to do the same thing, but perhaps in several different ways. Strategy lets the algorithm vary independently from clients that use it.



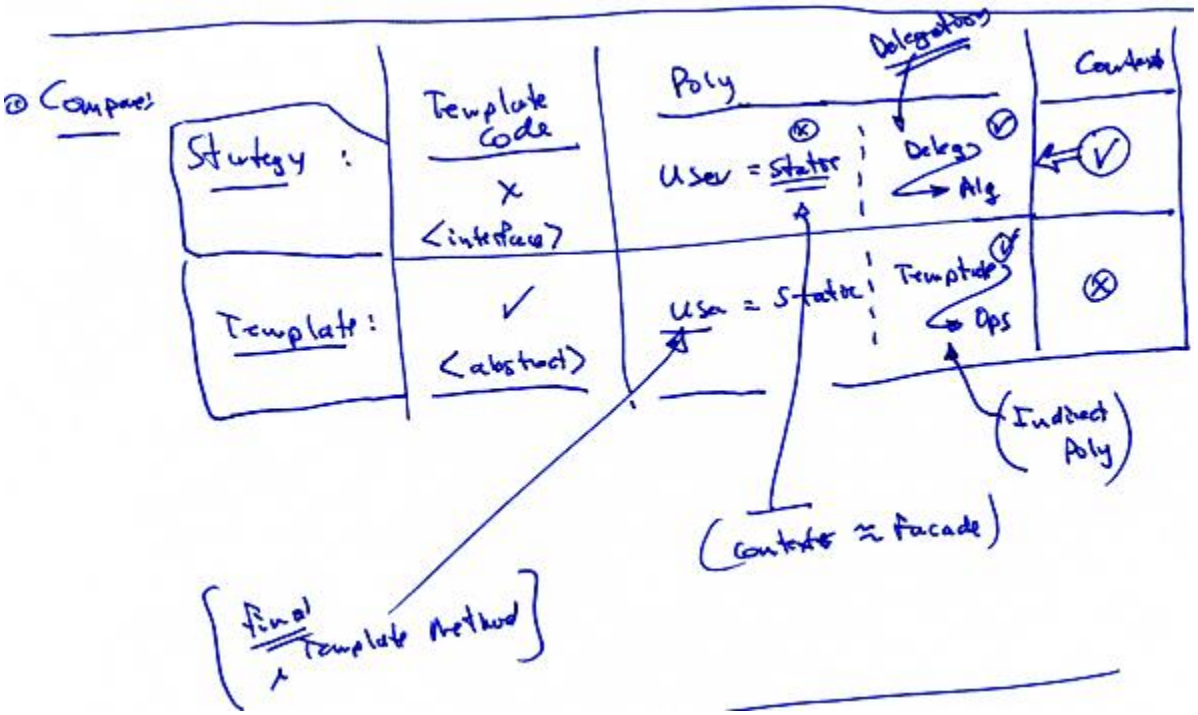
- Strategy pattern from GOF



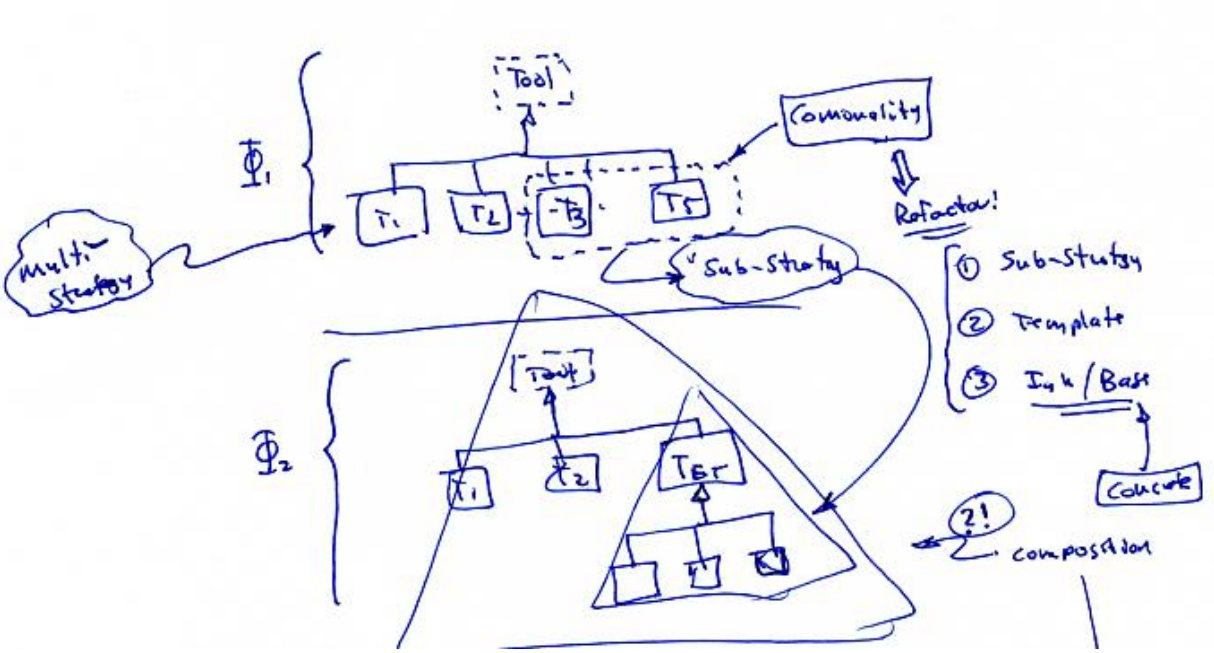
- In some cases a Strategy pattern can use refactoring, to eliminate common code in the concrete strategies



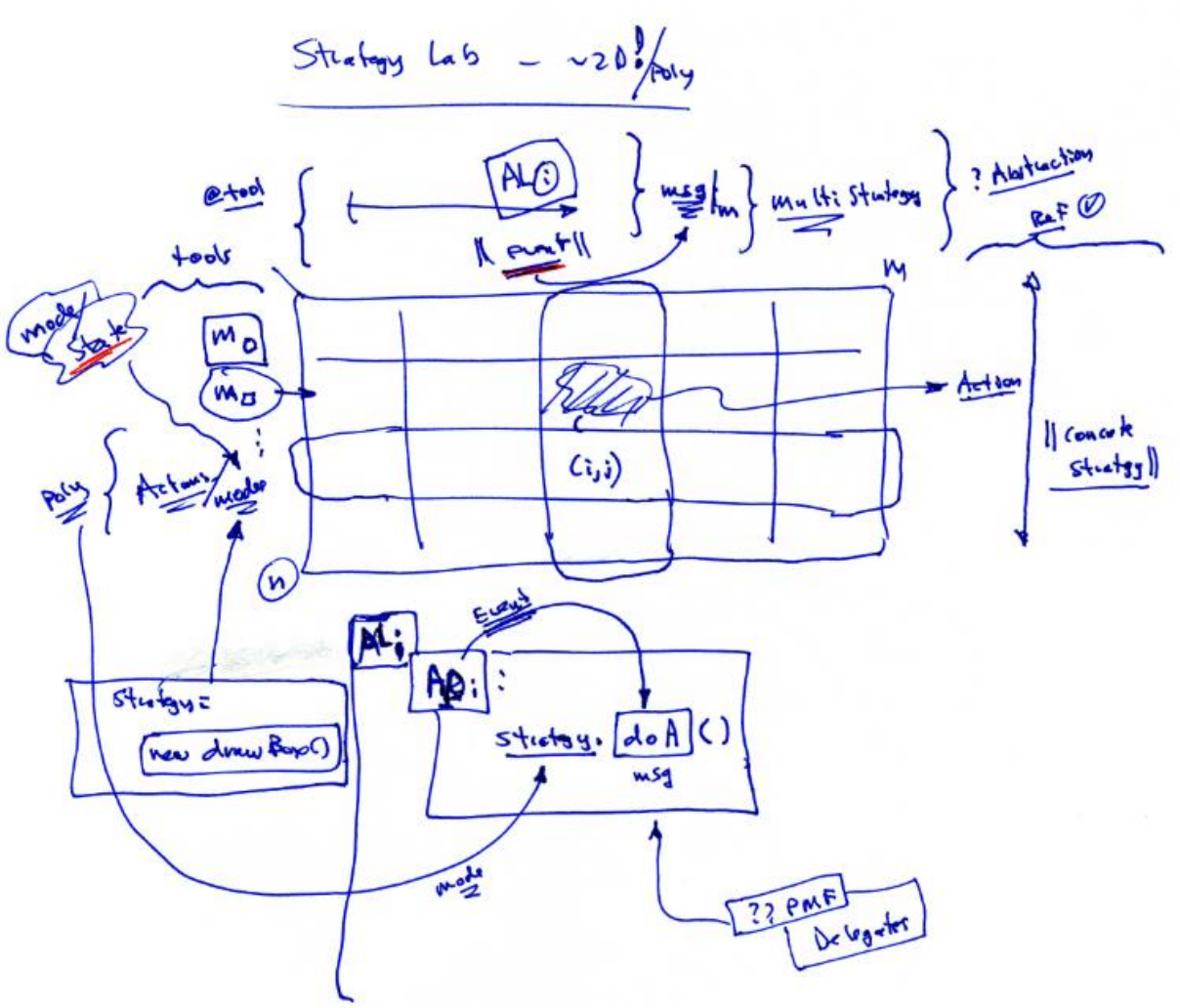
- This leads to the Template method pattern



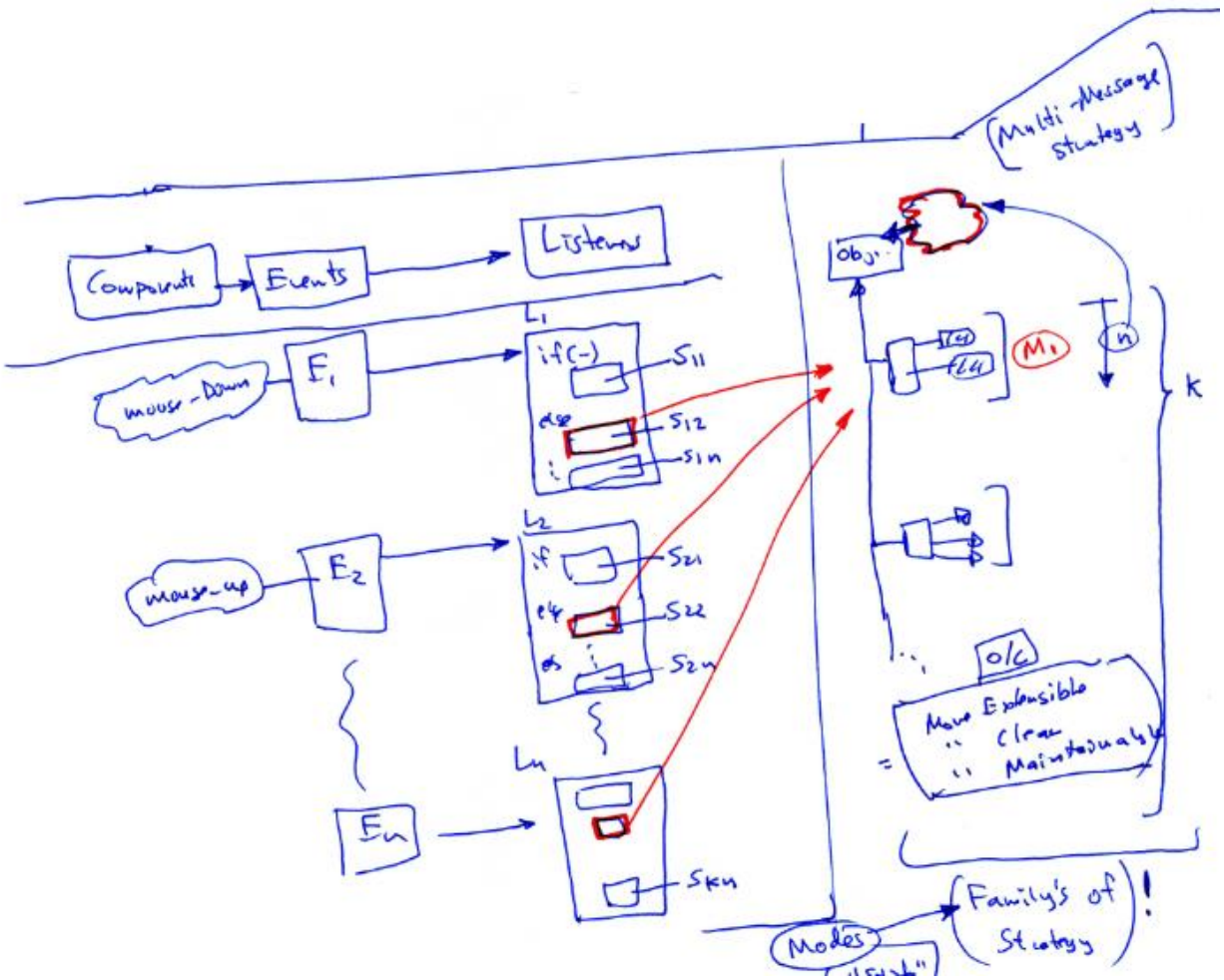
- Comparing the Strategy & Template patterns



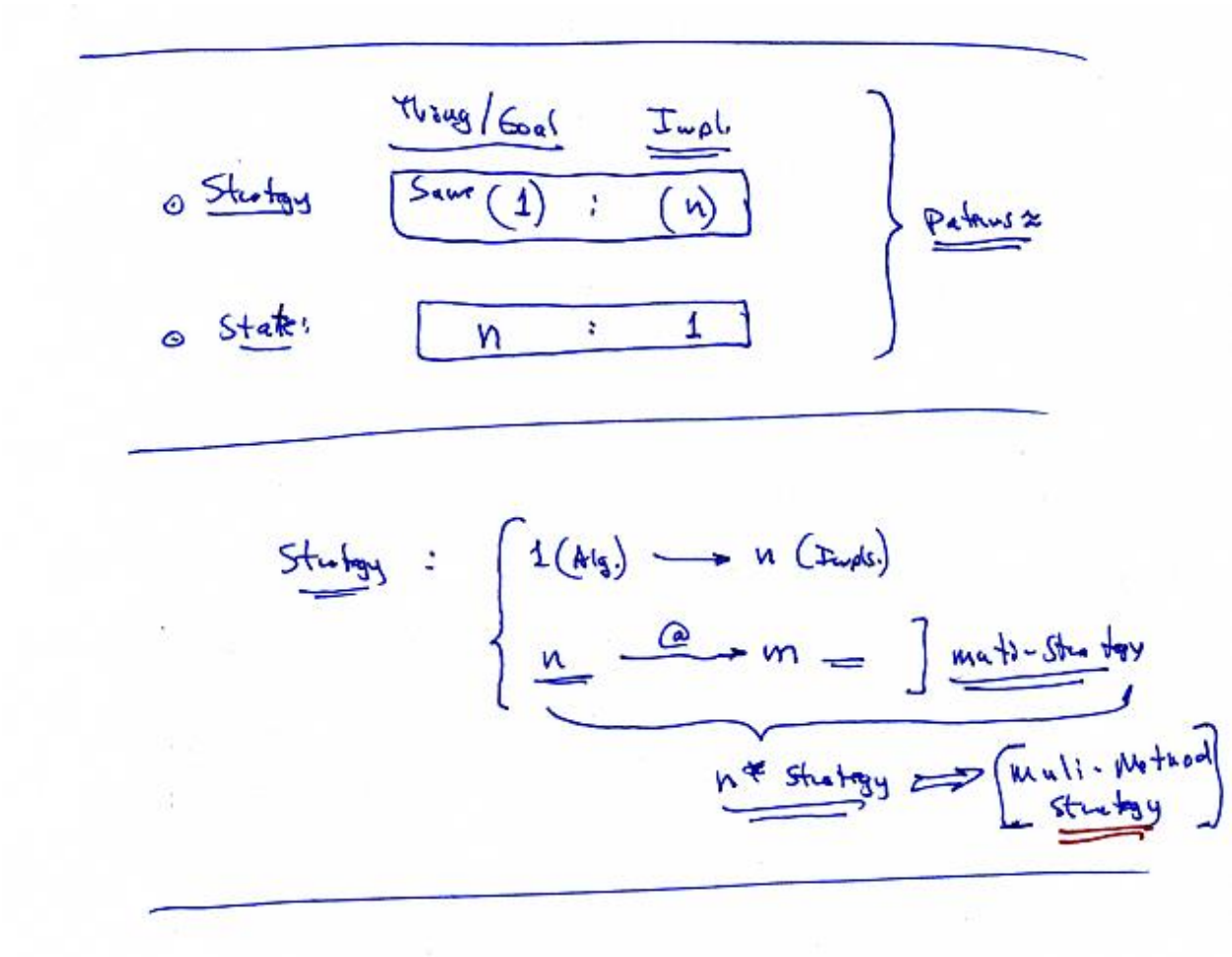
- Strategy Lab



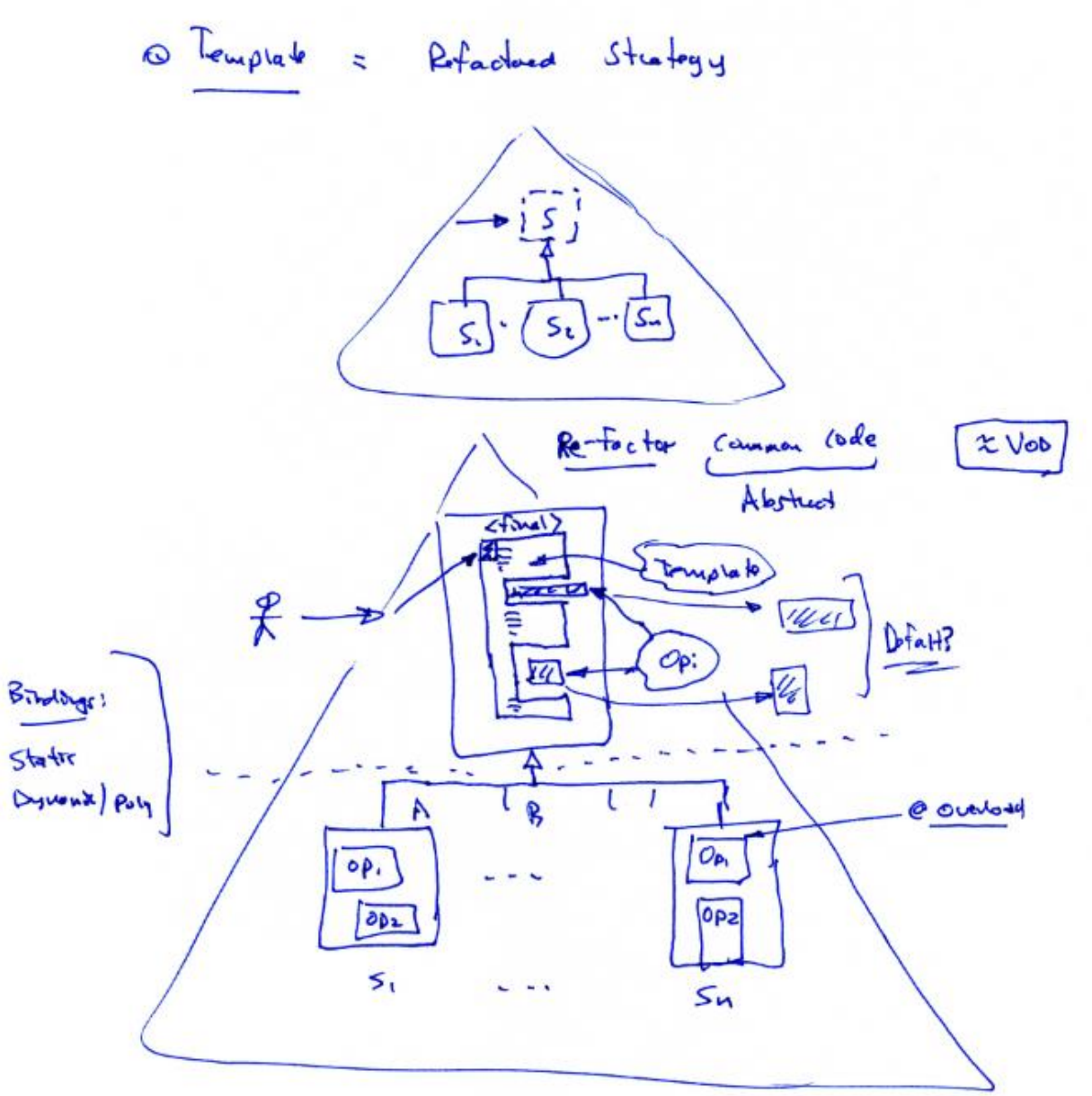
- Strategy Lab ≈ 2D Poly!



- Strategy lab is mostly just refactoring, moving code around



- Comparing state & strategy patterns



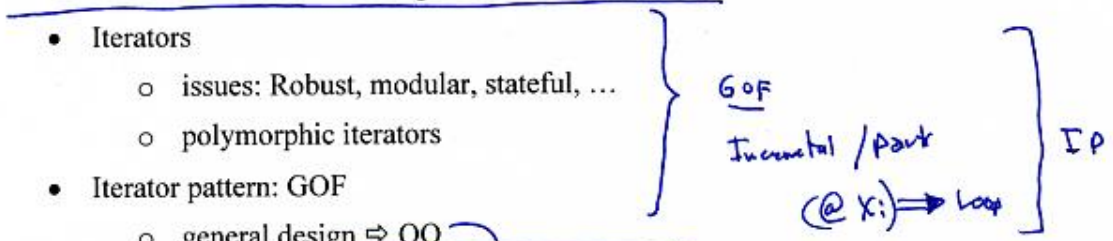
- Template ≈ refactored Strategy
- However it does not (typically) include a context – it has a different focus of improvement (the refactoring)

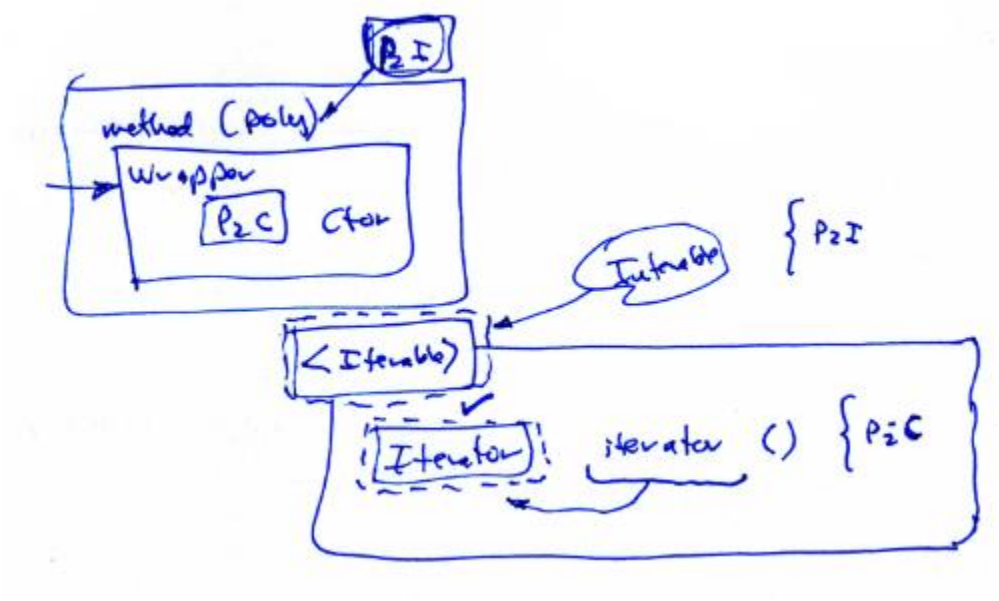
Day 5: Iterator Patterns.

Daily Topics

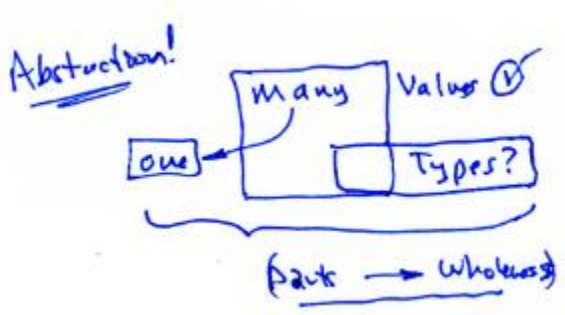
Day 5: Collections & Iteration

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

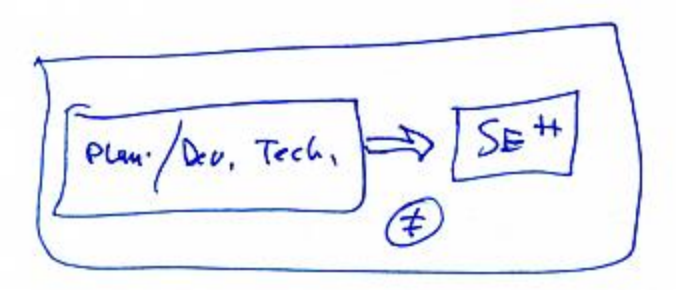




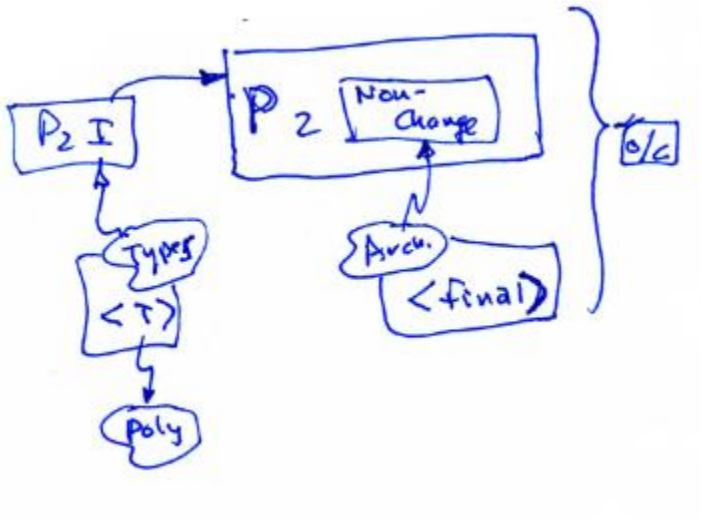
- Quiz & Review:
  - Factory methods encapsulate a P\2C constructor,
  - In a P2I method



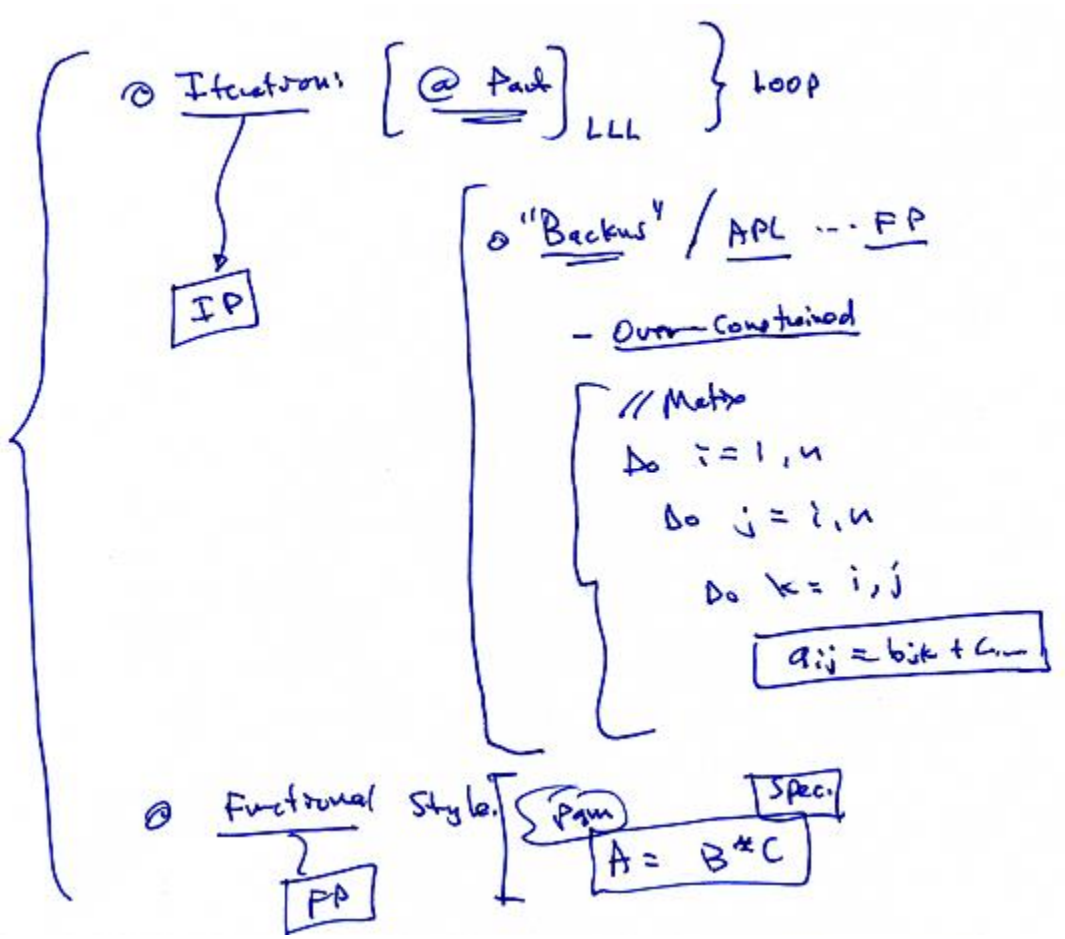
- Collections are an important abstraction;
  - Wholeness of parts;
    - Parts by values (elements)
    - And by types of elements (STA)



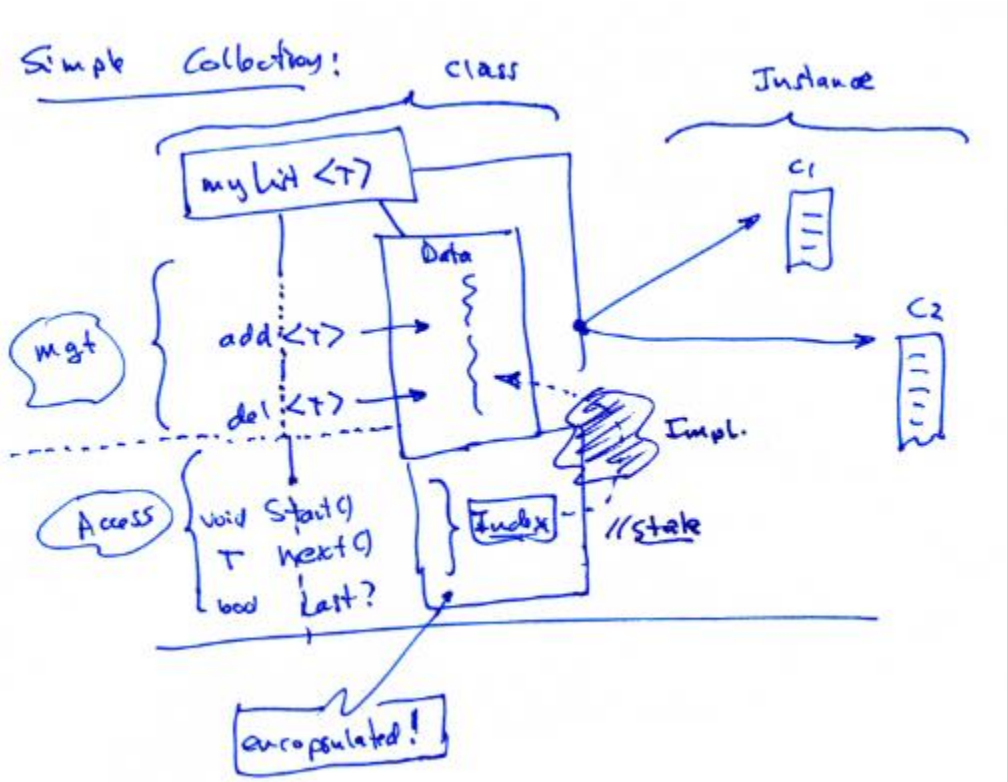
- Quiz: Reminder, Development techniques (Dynamic binding, modularity, Poly) are not the same as SE benefits (extensibility, open/closed, …), which should each ⇨ $$$ business benefits.



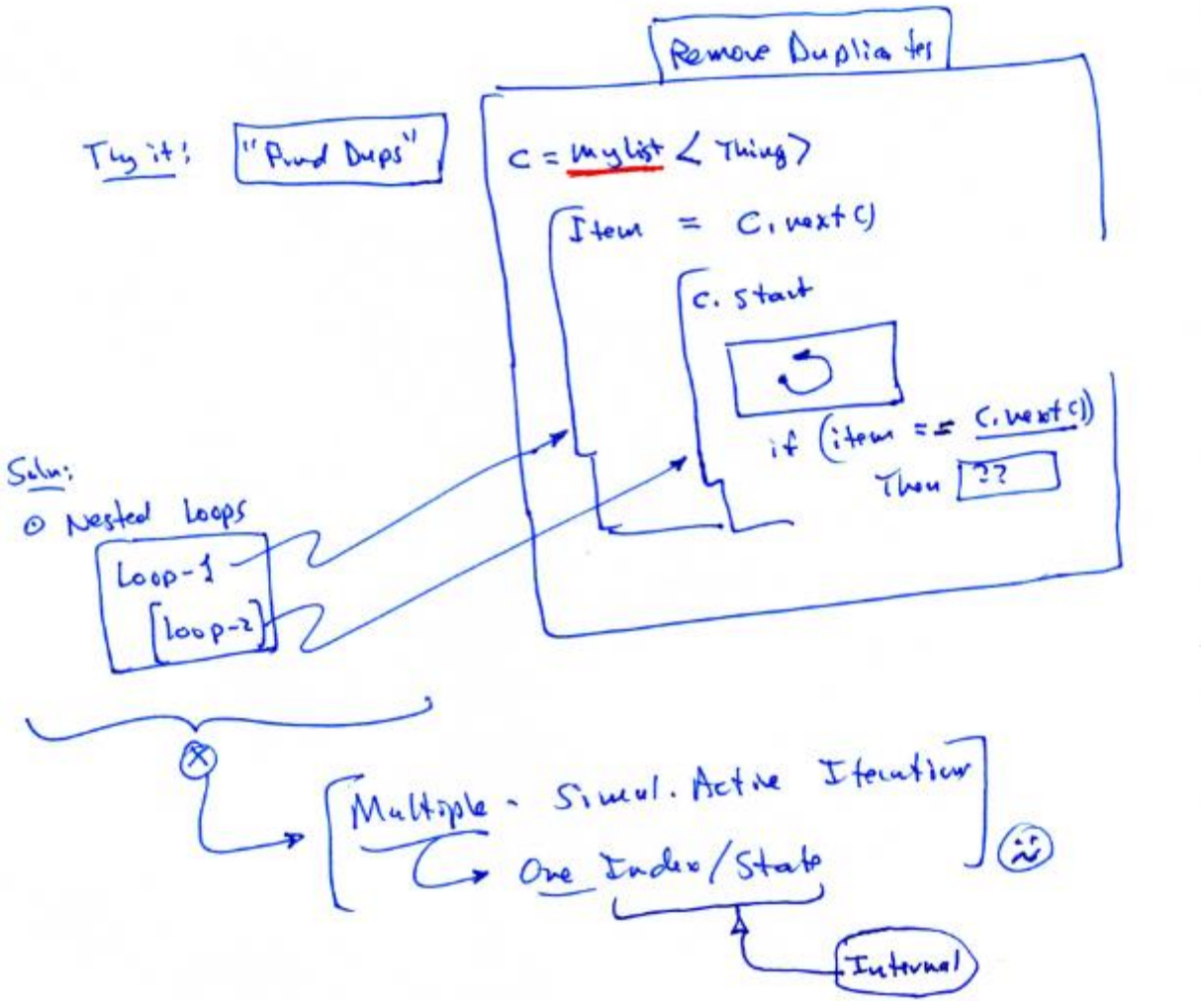
- There is an additional application of Open/Closed principle;
- Poly is the main one we have studied; *fixed interface*, varying poly concrete sub-types
- Another would be architectural – making a method <final> (*static binding*), so that it cannot be changed, but it can delegate to a (P_2 I) reference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



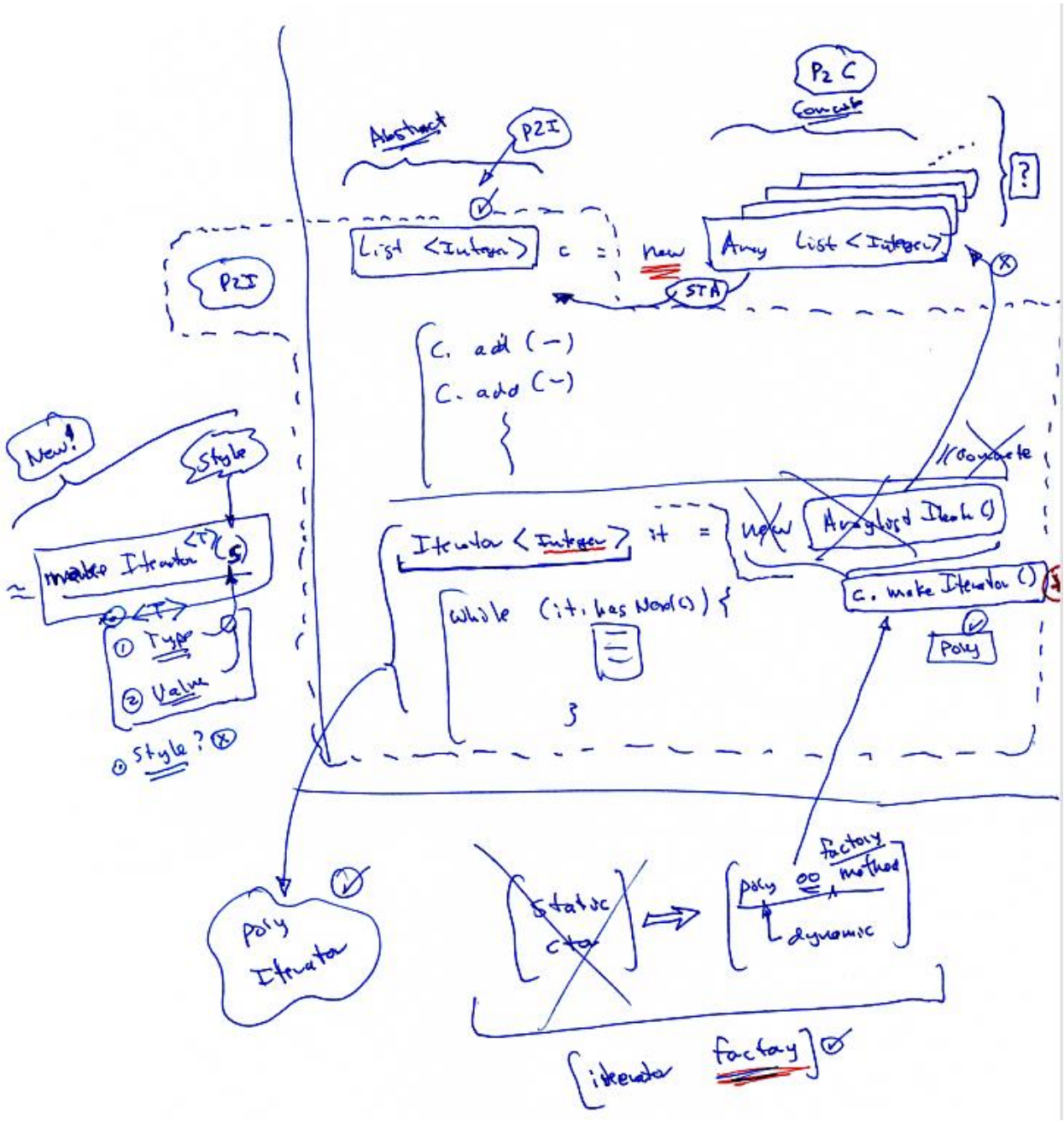
- Iteration in IP languages is too *low-level* (semantics)
- ⇨ *over constrained*



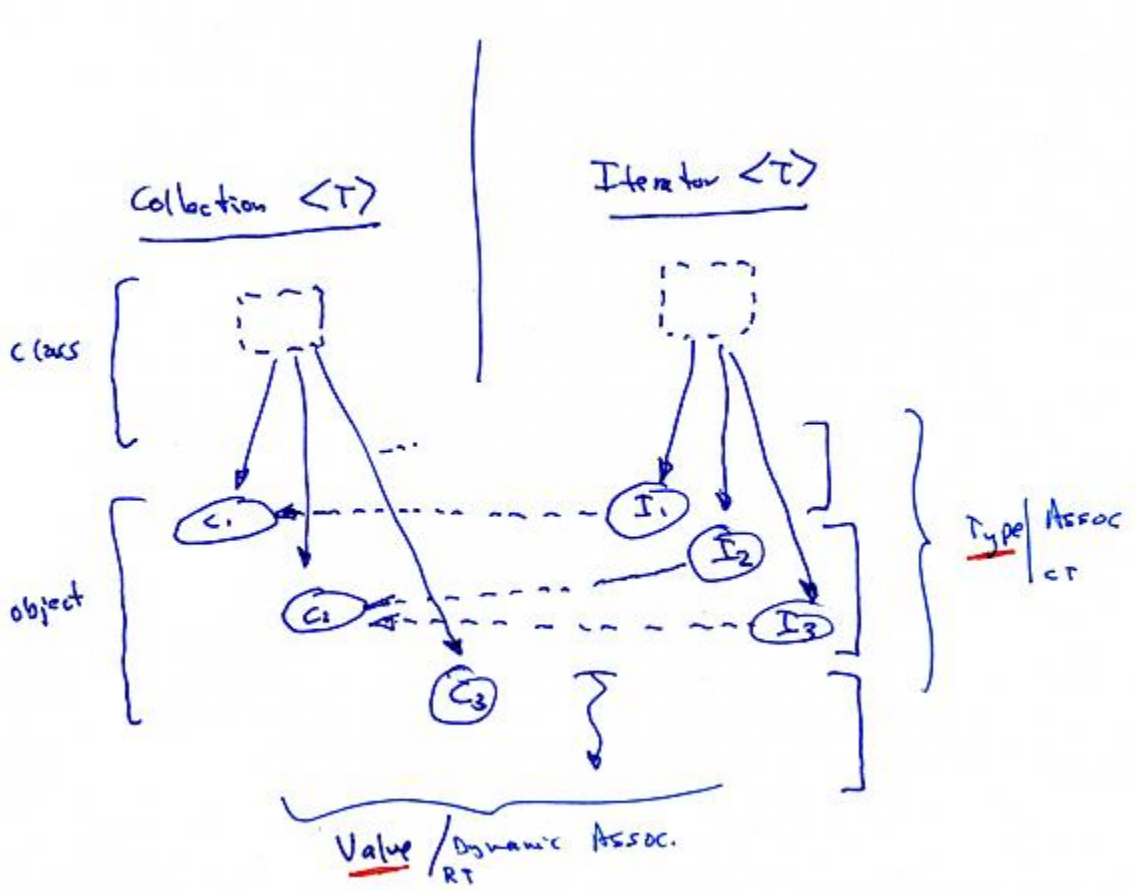
- Example of a simple collection implementation
- With some access (iteration) added



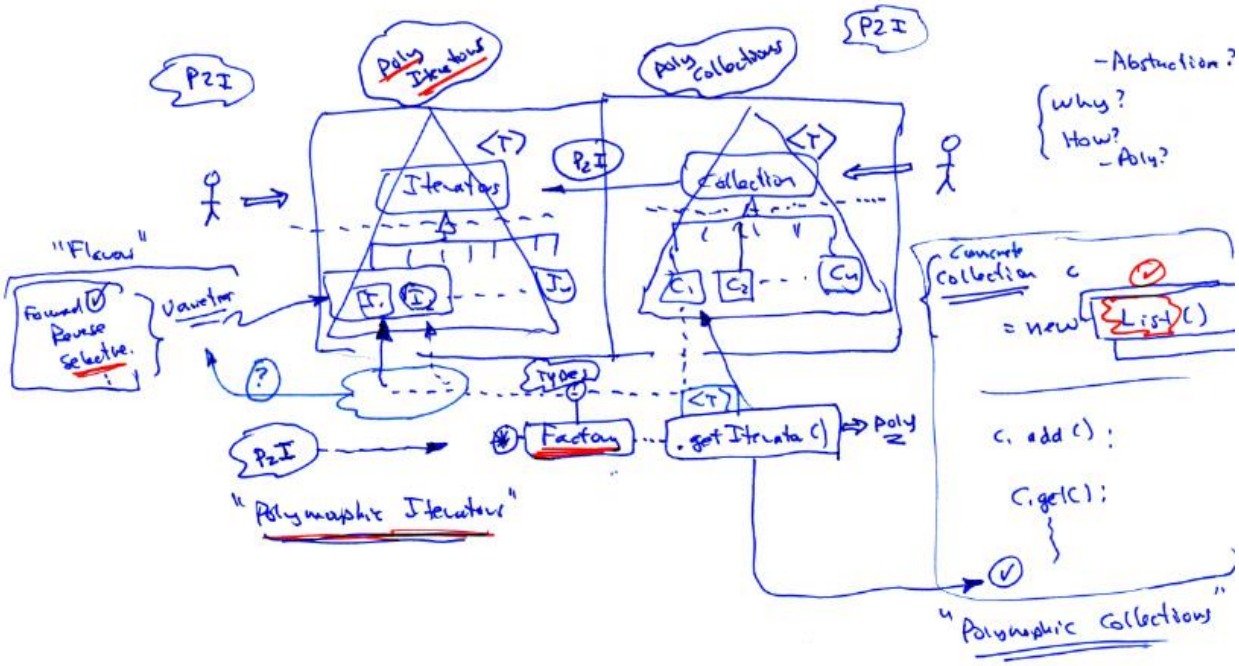
- Testing it – fails!



- Using P2I for iteration



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

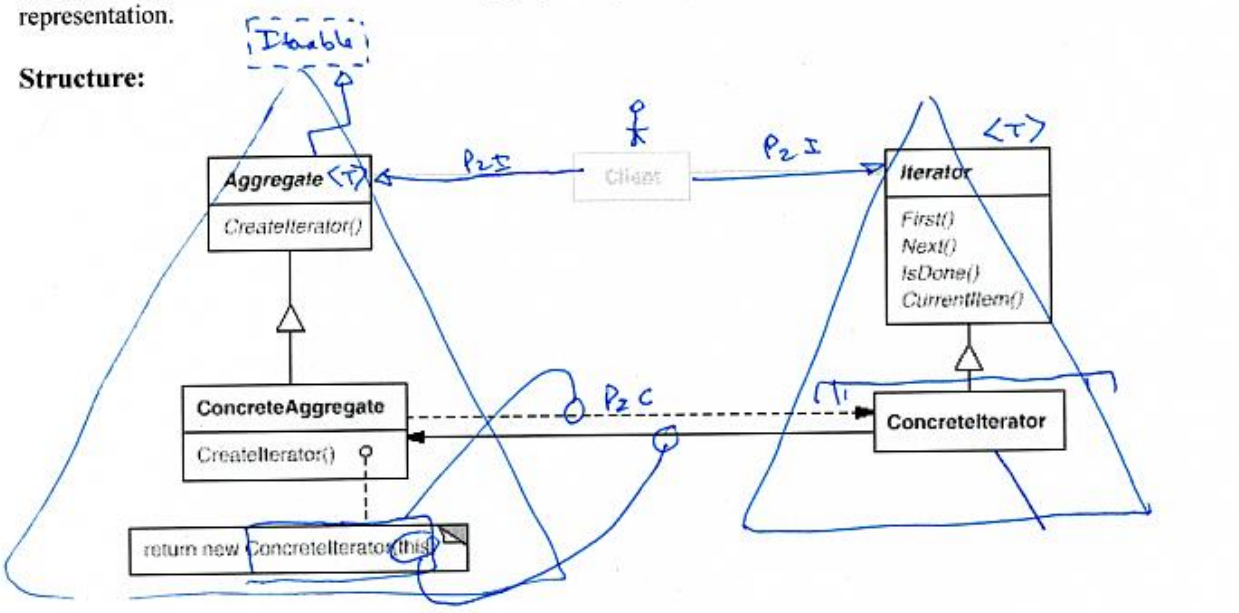


- Dual hierarchies of iterators and collections
- Related by type, and value

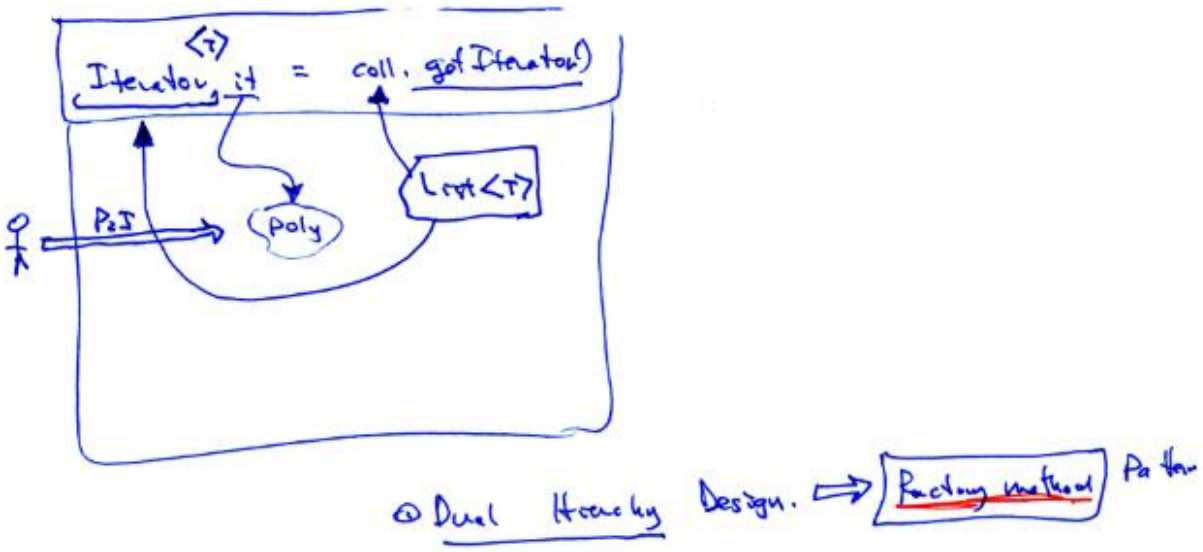
Iterator Pattern

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

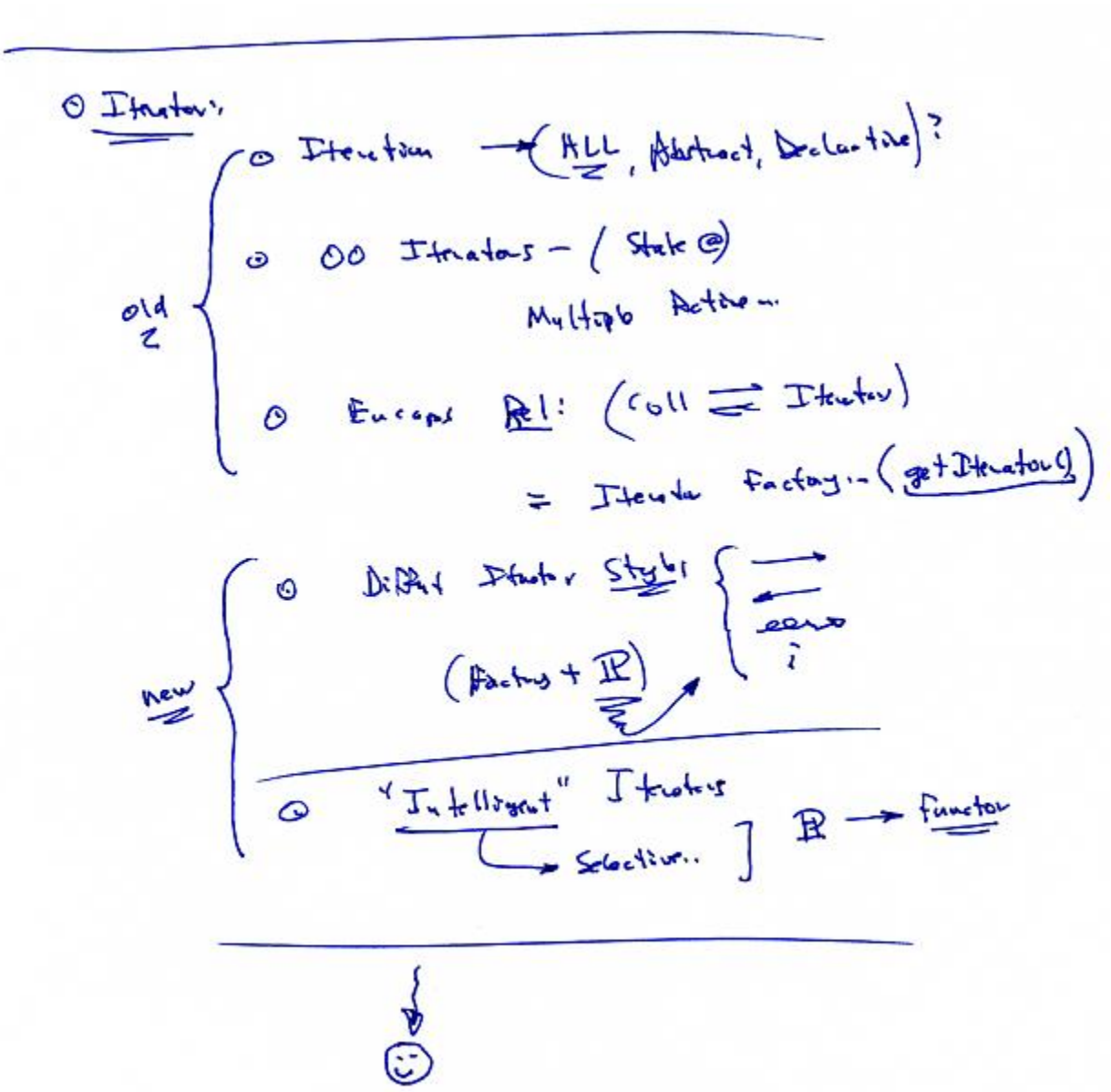
Structure:



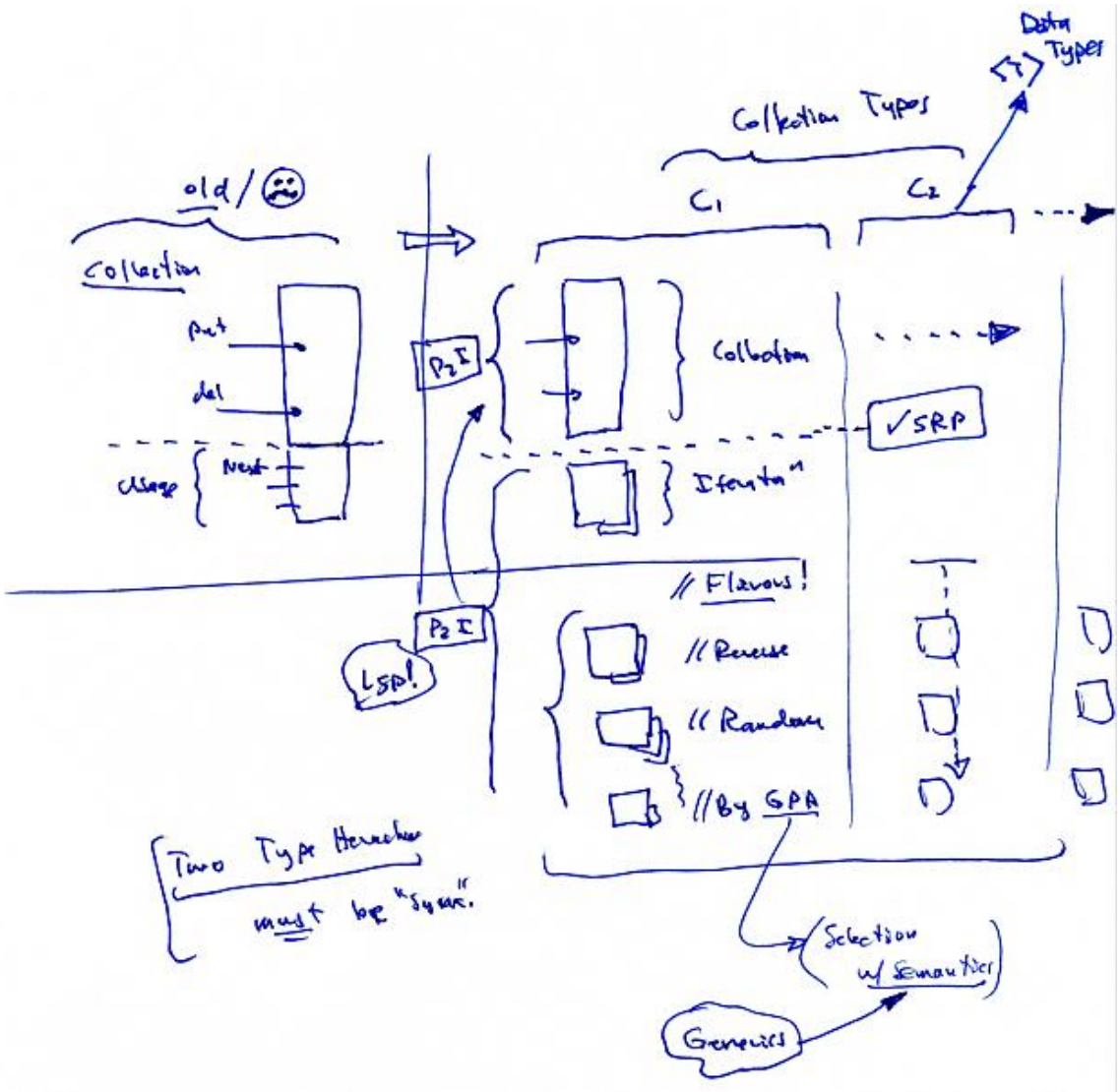
- GOF Description



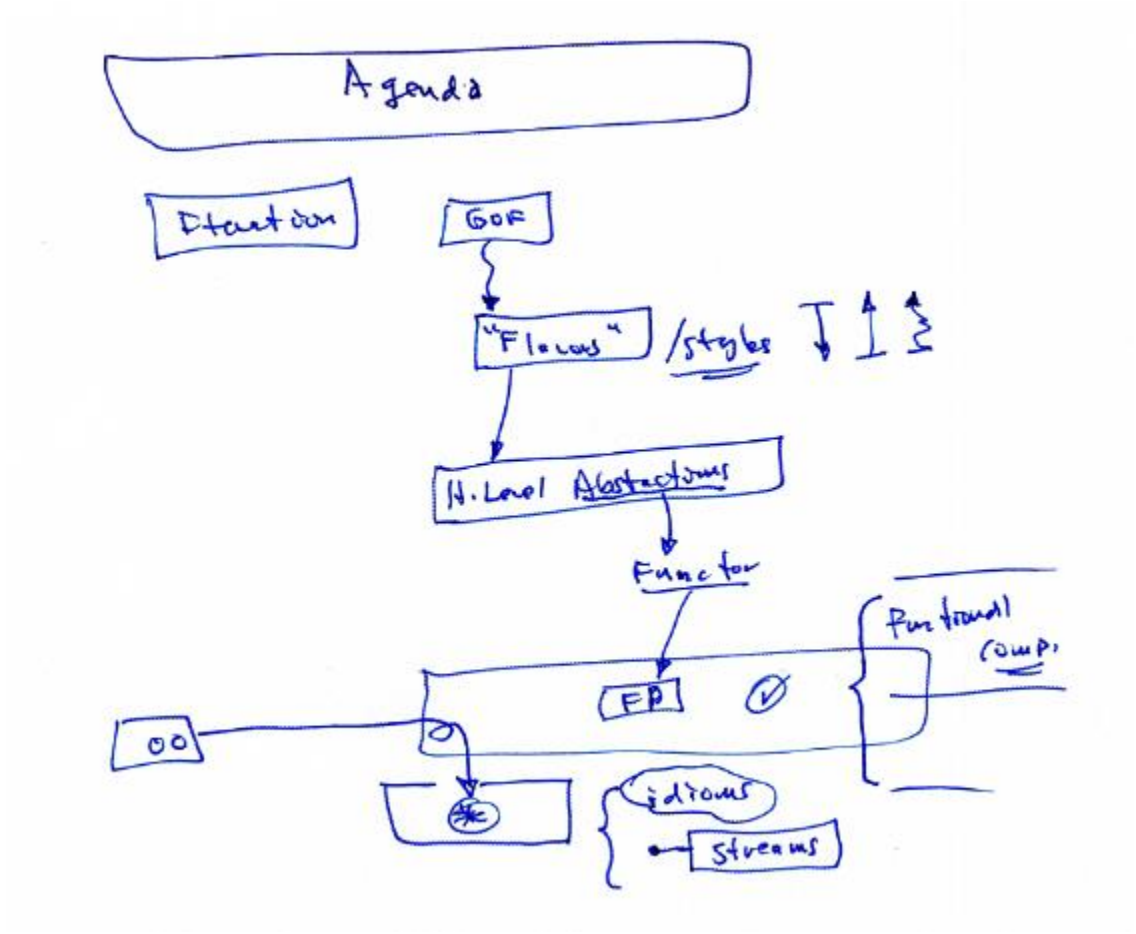
- Factory method for "*getIterator()*" instead of static constructor – concrete type



- 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:

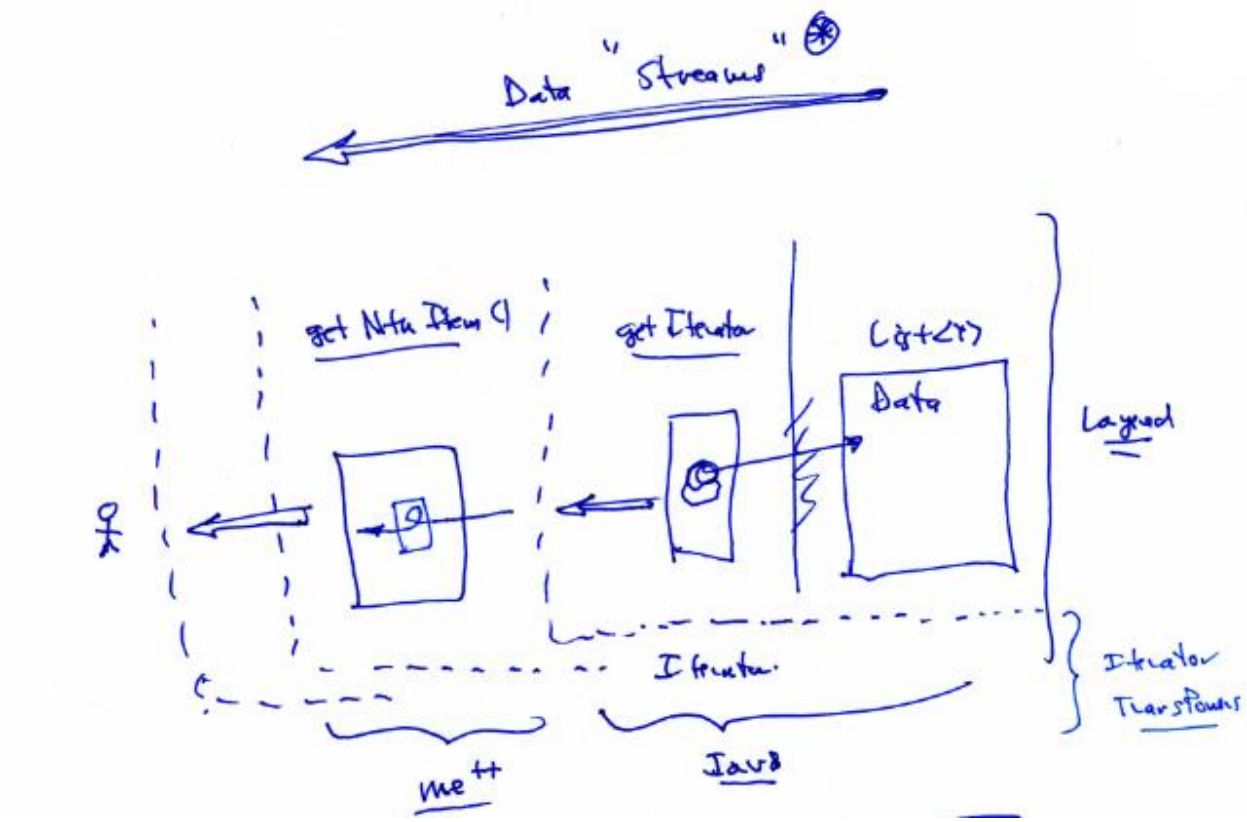
Iterable

	Complete Traversal	Selective Traversal
External	Iterator<T> it = c.iterator(); while (it.hasNext()) ProcFun(it.next()) Or, *foreach* (T e : c) { … }	Iterator<T> it = c.iterator(Pred p); [Then use *normally*! ??]
Internal	c.doAll(ProcFun)	c.doAll(Predicate, ProcFun)

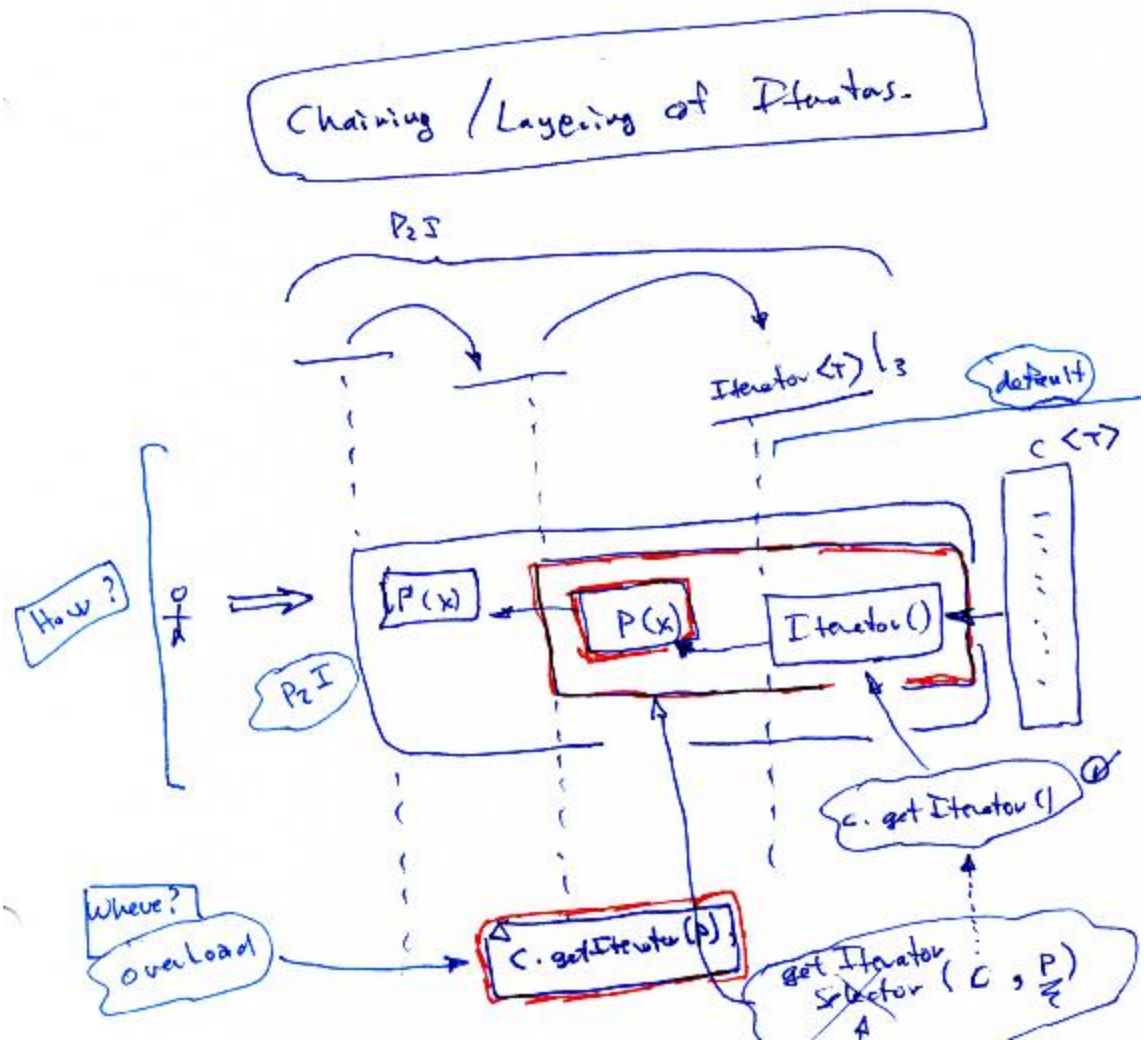
old

new

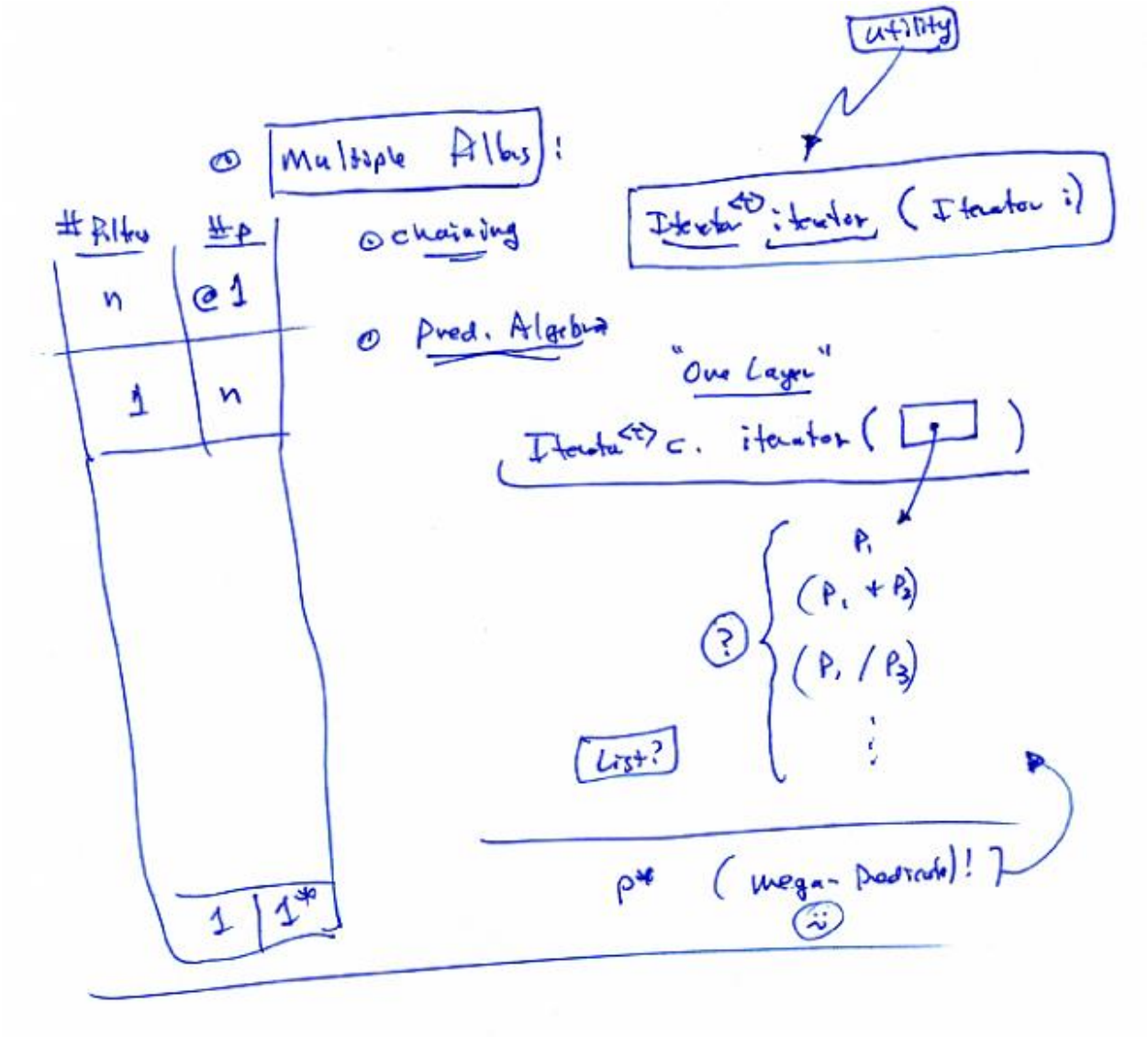
- New iterator types



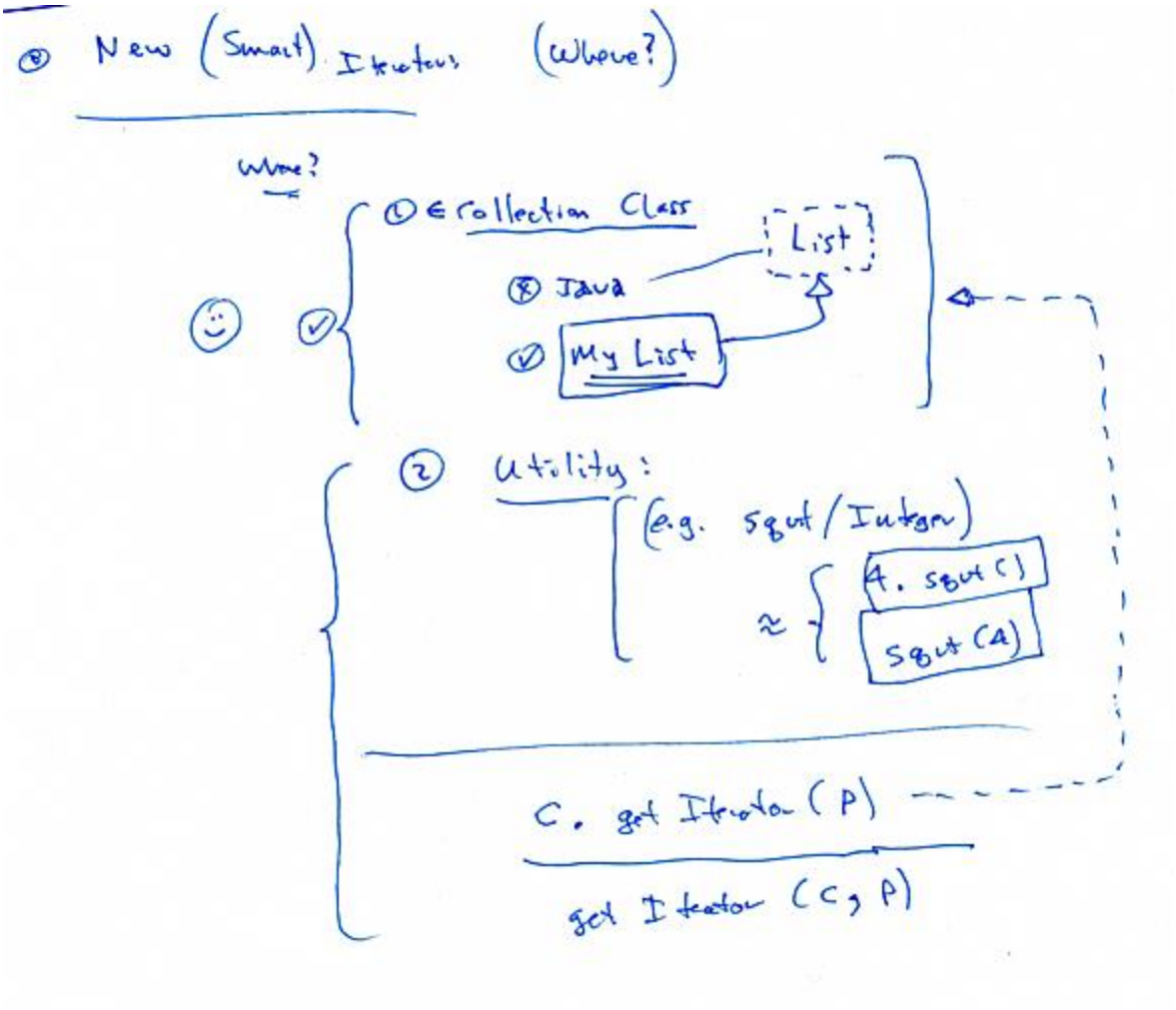
- Chaining of iterators by levels of P2I gives a new design approach
- *Data streams* (soon in Java-8!)



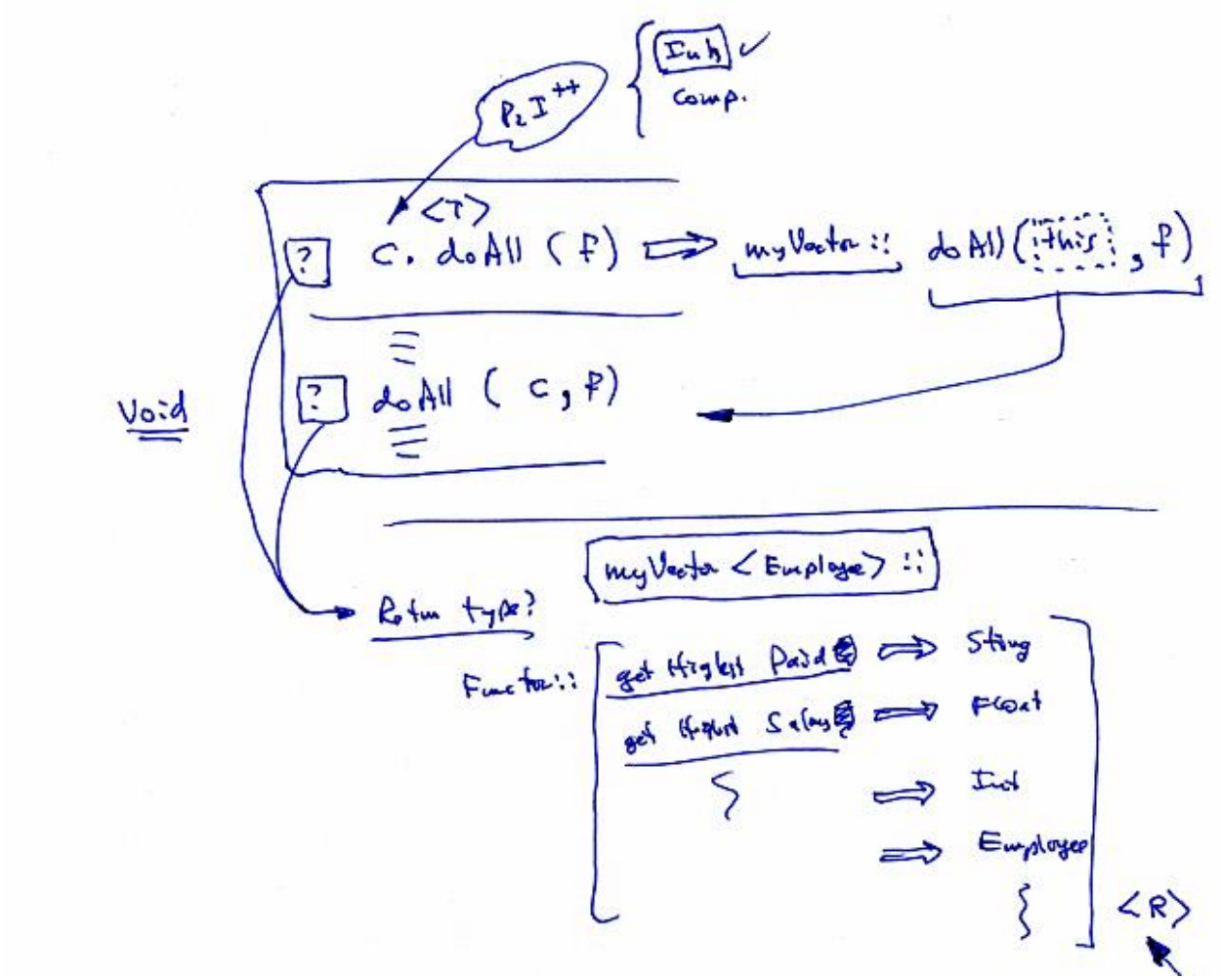
- Now we know what to do, how to implement it?



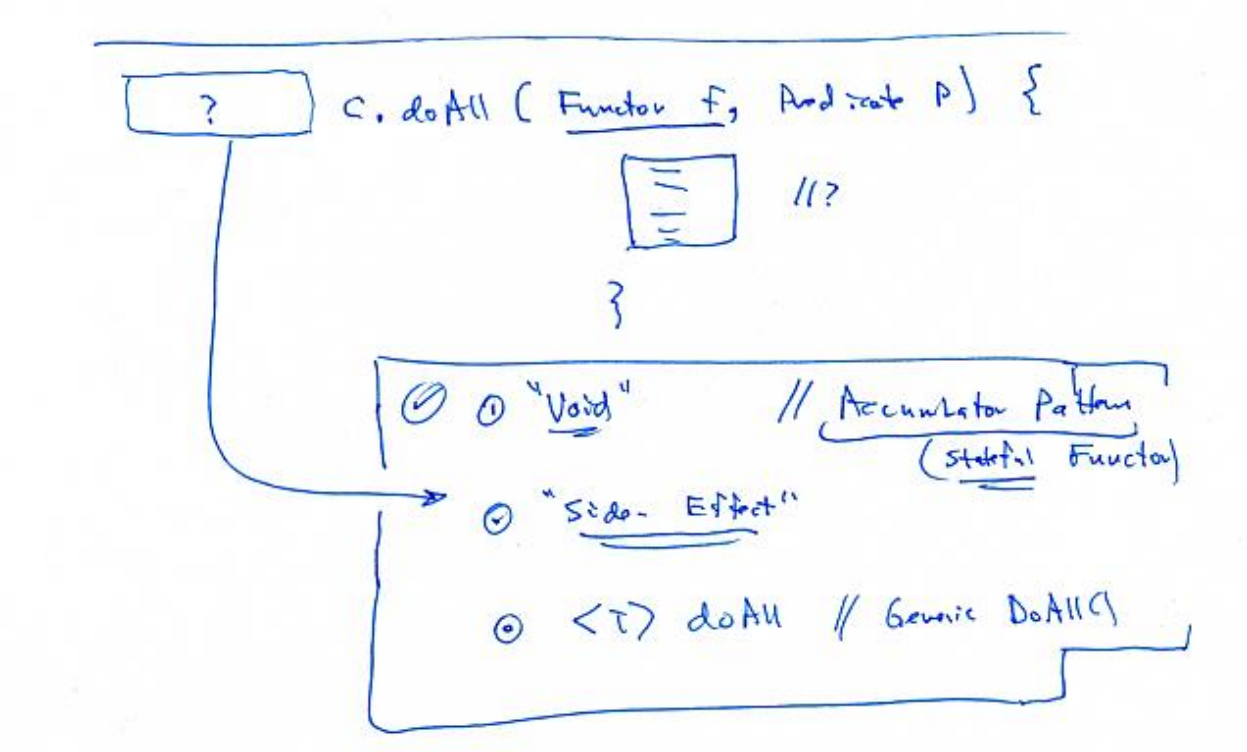
- There are multiple ways to do this;
  - One is to chain multiple iterators,
  - Another is to have a single filter with a compound predicate
  - Another would be to have one "*mega-predicate*" with multiple logic selections- in it, but that is a poor design…



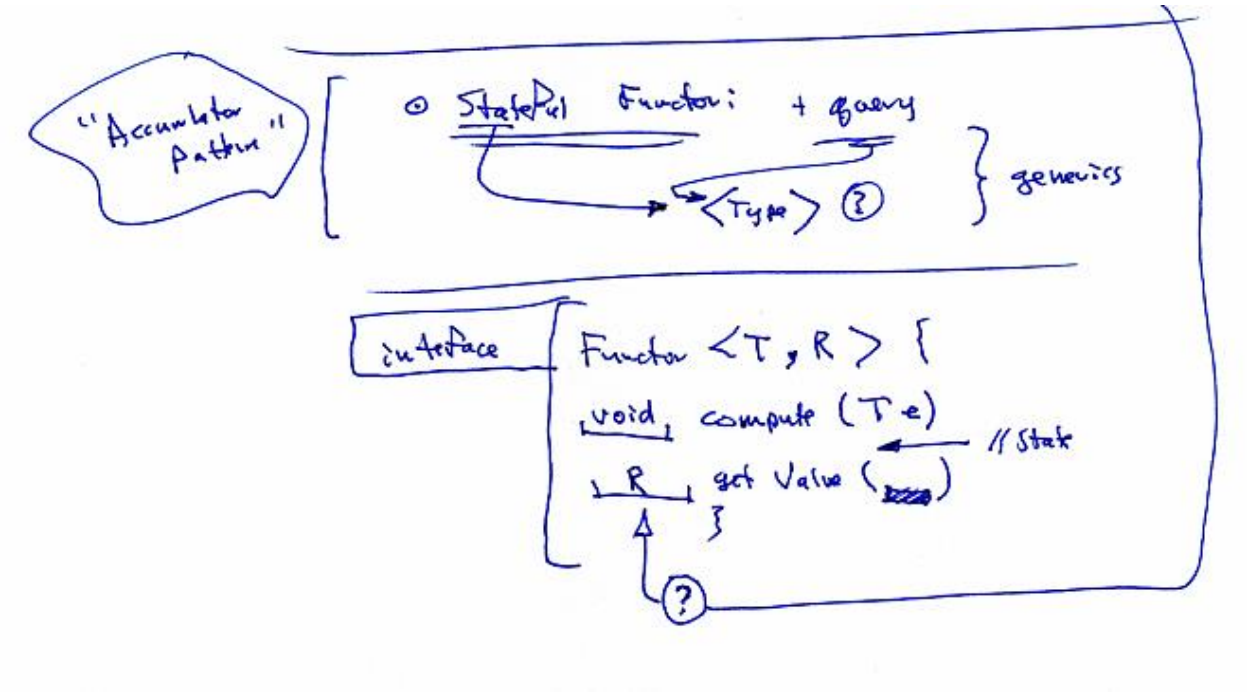
- Where should the new methods be implemented?
- In the collection class (we don't own it), or as separate utilities?



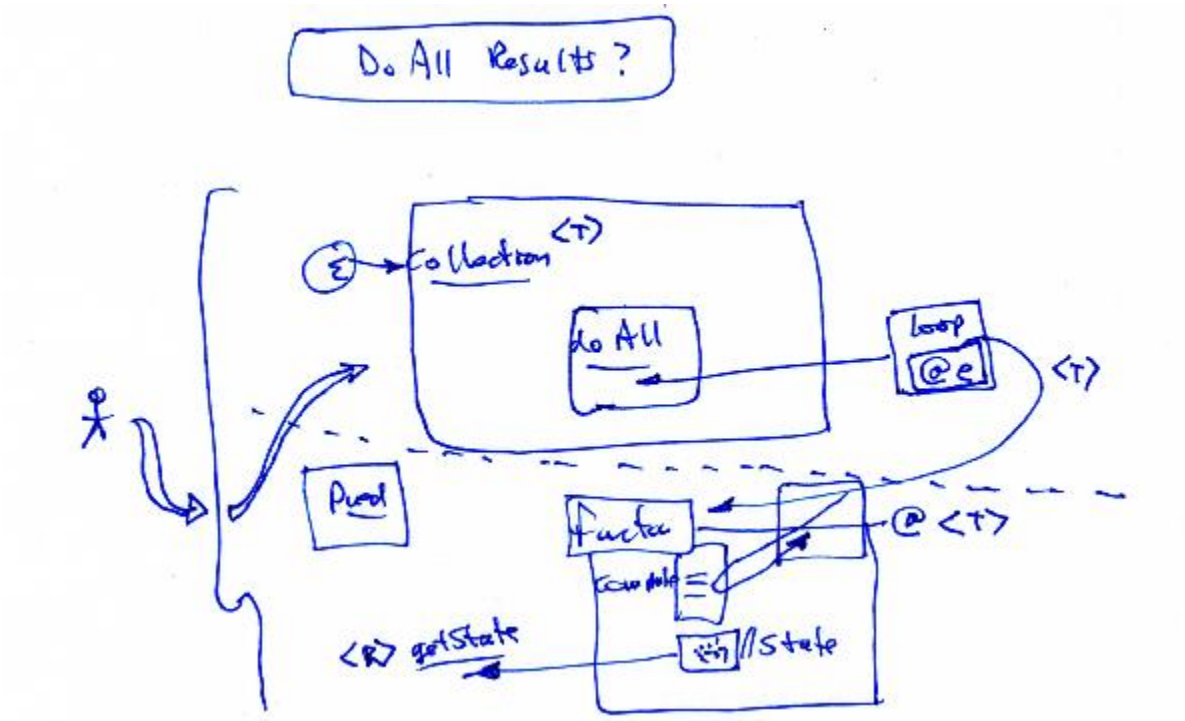
- Other design issues – return type of *doAll*?
- Functor is element-by-element
- Never sees context (= collection)
- *doAll()* uses it for a collection, how to return results?



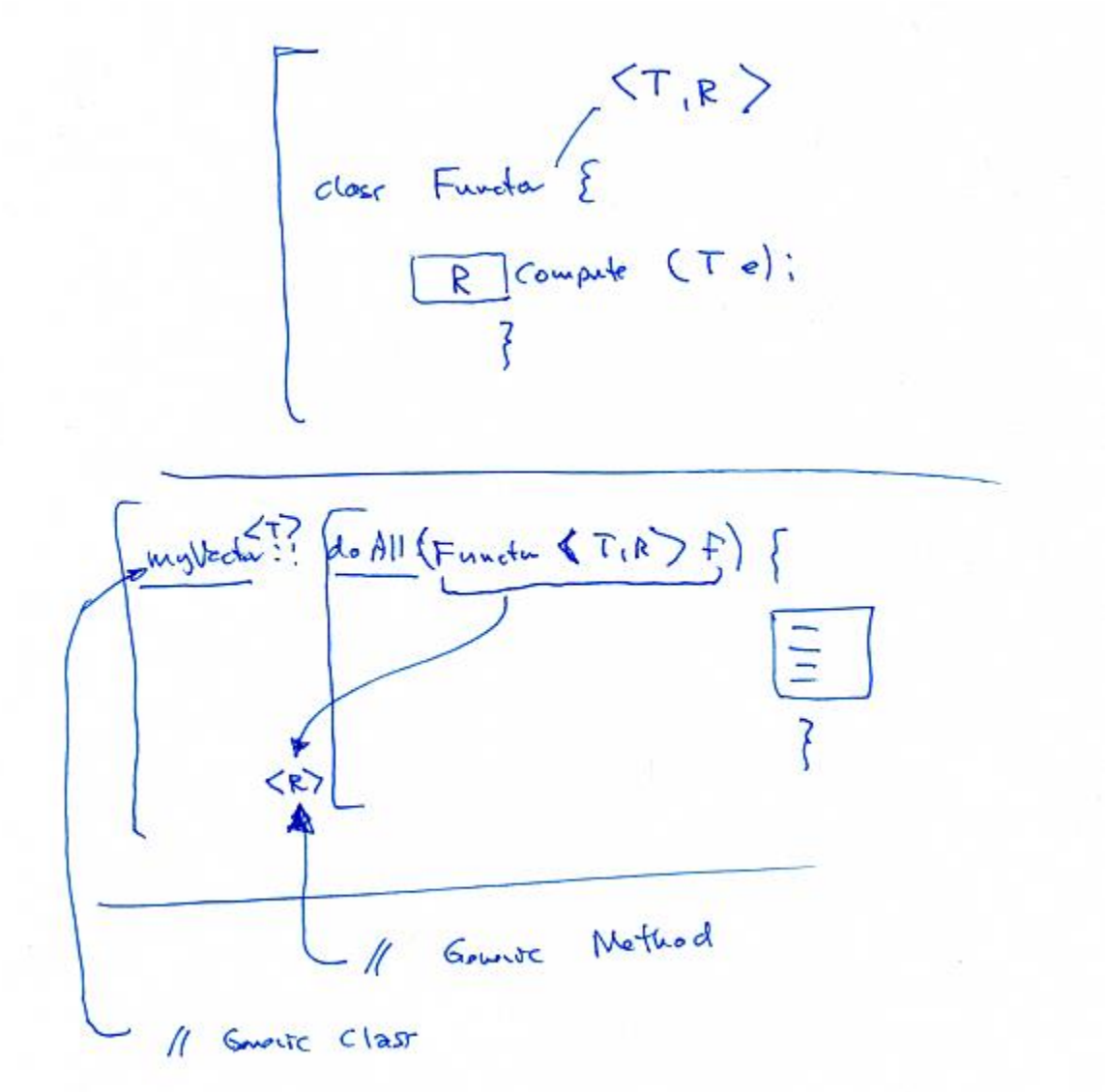
- three options:



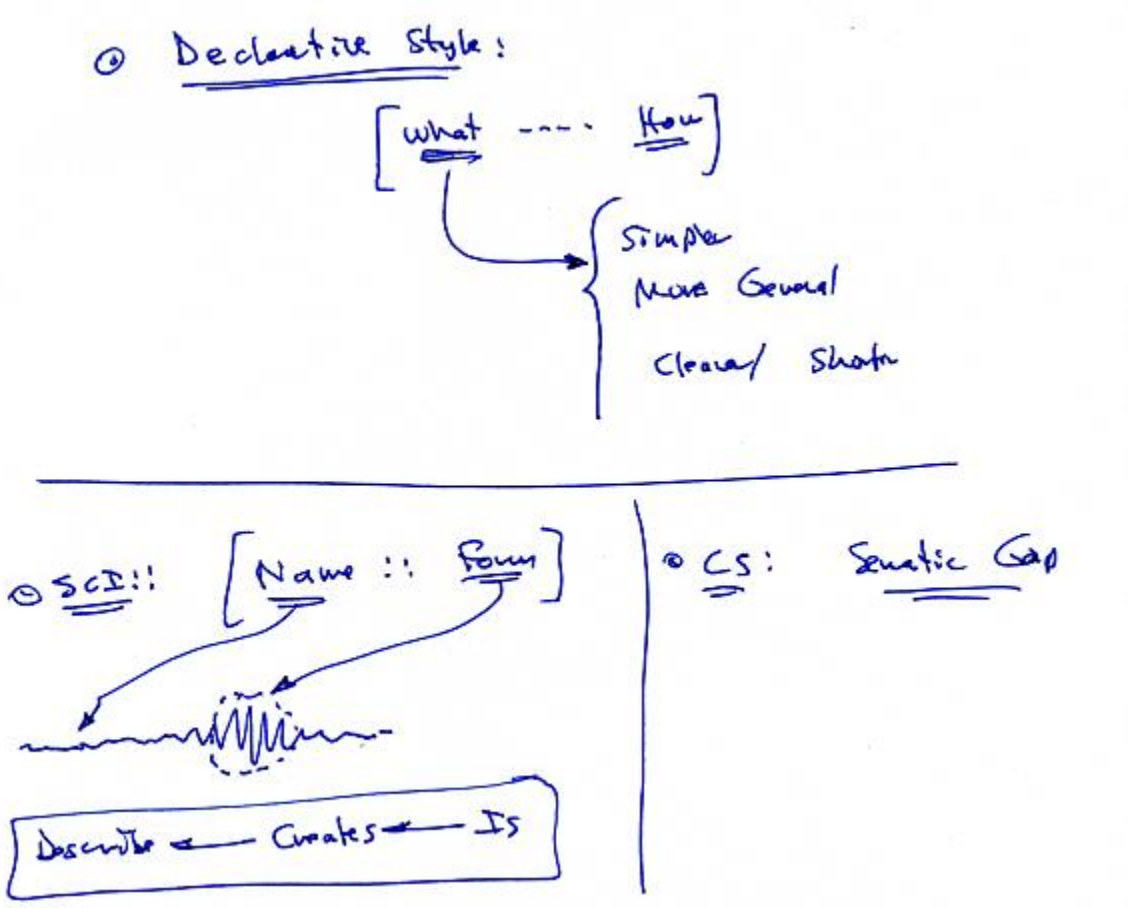
- We will use the accumulator pattern, a stateful functor



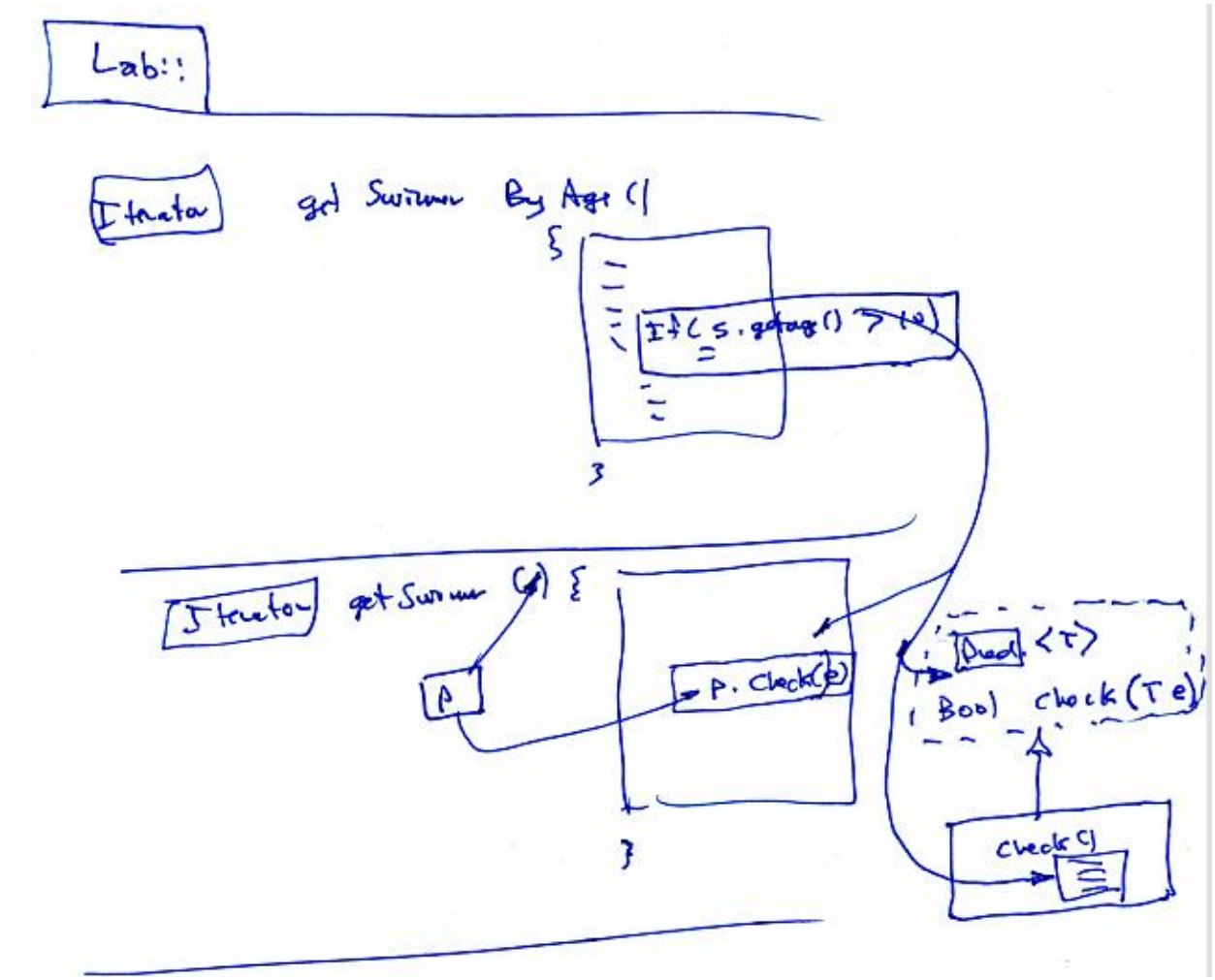
- doAll return via stateful functor



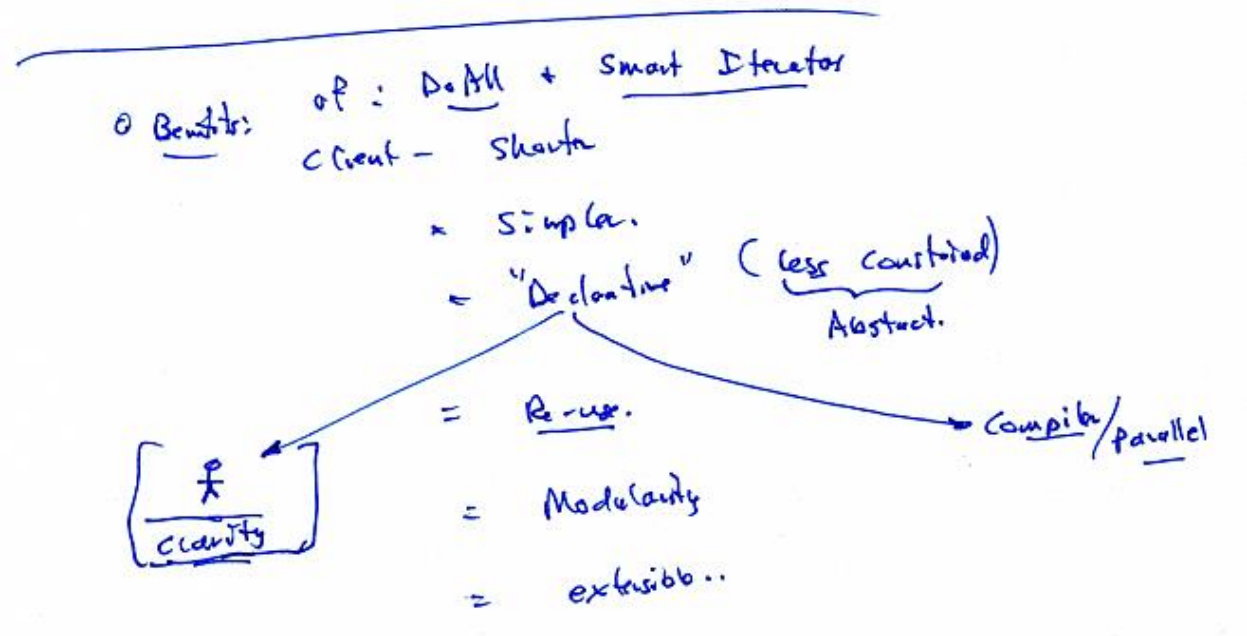
- An alternative would be a generic *doAll()* method, different return types each time, depending on the argument types!



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



- 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* ⇨ *MyVector*
  - *accumulator* pattern
    (or, *generic methods*)
- Iterators…
  - Selective iterator
  - make ***Iterable*** → *views*
  - add compound predicates { composition… }
- FP style → **SE**$^{++}$ (name: foun)
  - Declarative; more abstract ⇨ simpler, shorter, clearer, …
  - *Loopless* programming Metrics…
- Standard FP functions (idioms…)
  - Filter, map, reduce } List comprehensions | Intrusive!
  - Reduce ⇨ foldr, foldl, …
- Exercise/Lab
  - add selective to *myVector*
  - then: selective & internal
- Lab:
  - Iterator Lab'

- FP ~> IP { … (JOOQL, …) Guava Scala
- Constrained Generics.

Day 7a: More Iterators & FP

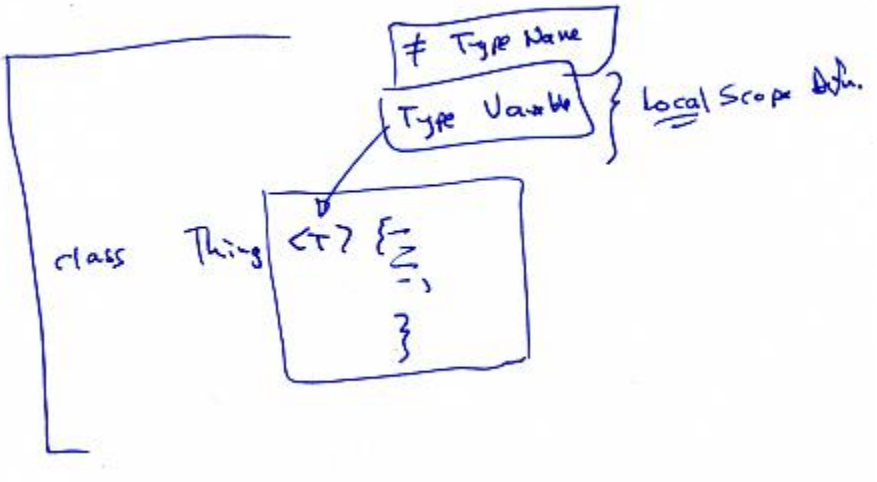
Daily Topics

Day 6a: **Functional Iterators…**

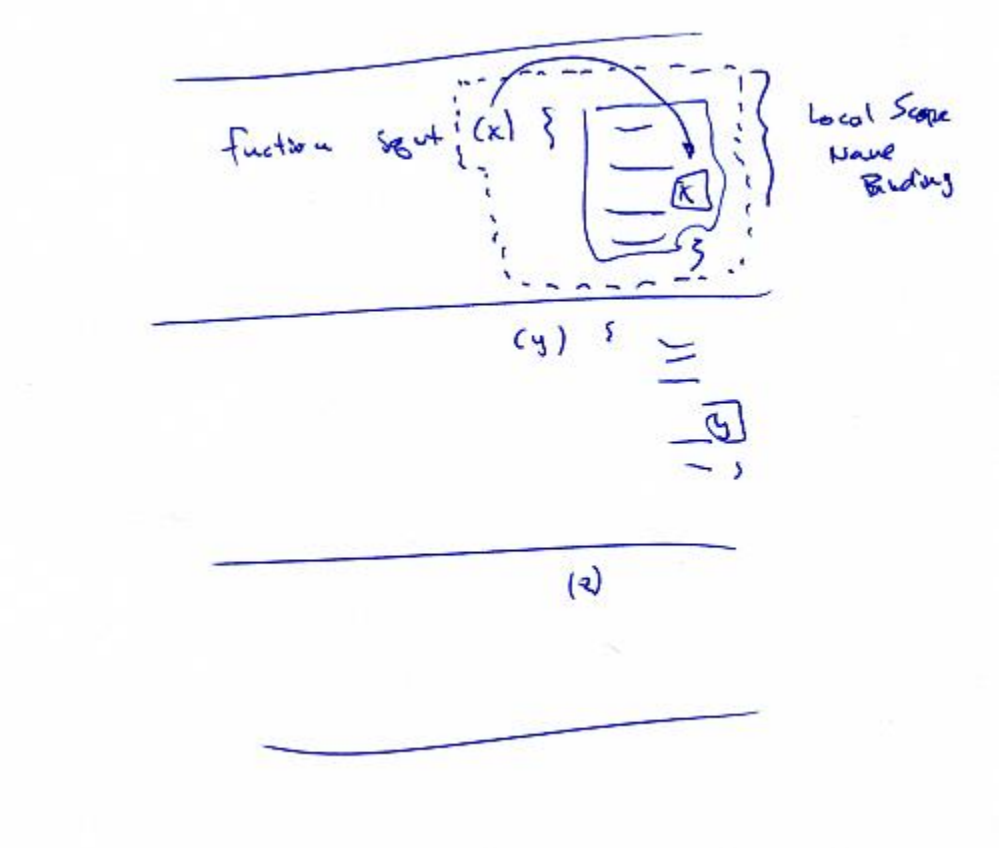
- Iterator Exercise Review
- Iterators…
  - make ***Iterable*** → *views*
  - add compound predicates
- FP style → **SE**$^{++}$
  - Declarative; more abstract ⇨ simpler, shorter, clearer, …
  - *Loopless* programming
- Examples…
  - Guava iterators [GNU]
- Generators

Layered Iterators
Predicate Algebra (DSL)

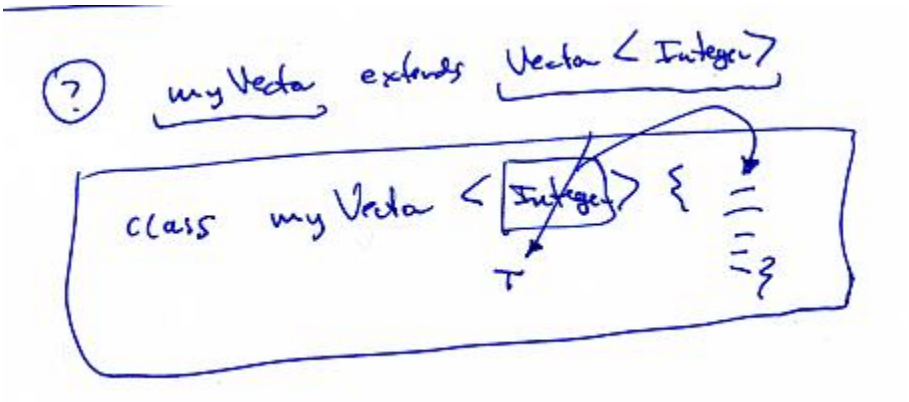
Review of Generics::



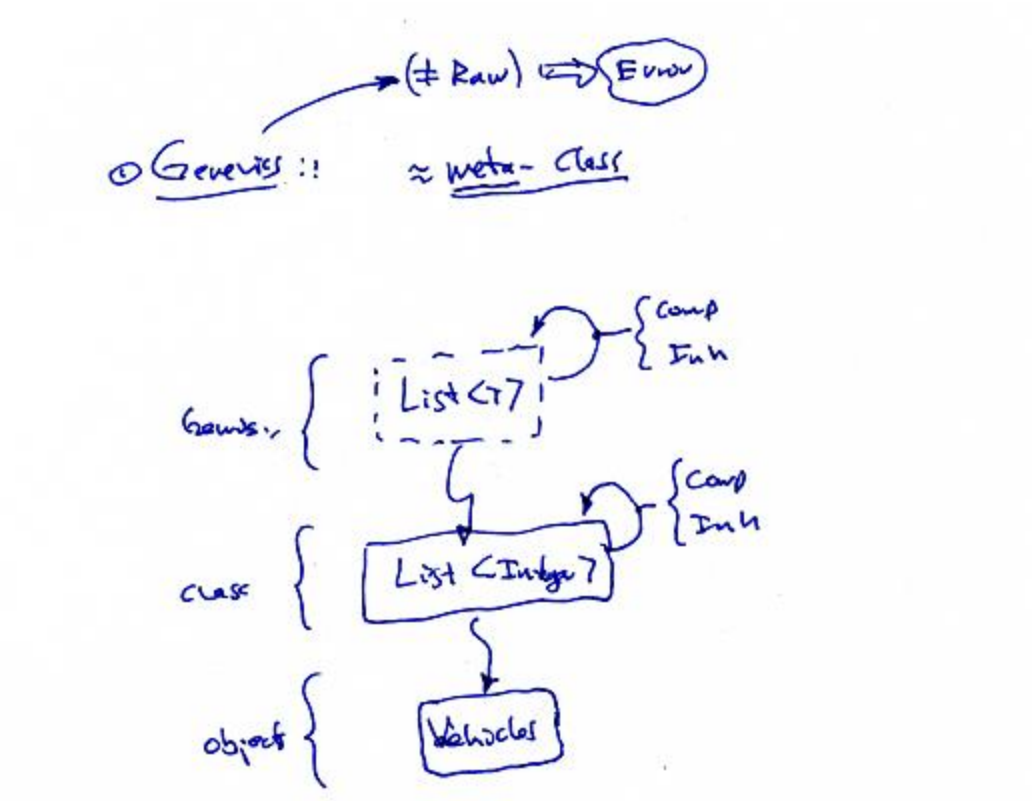
- Generics are type abstractions
- The scope of the type declaration (<T>) is the following body of the class



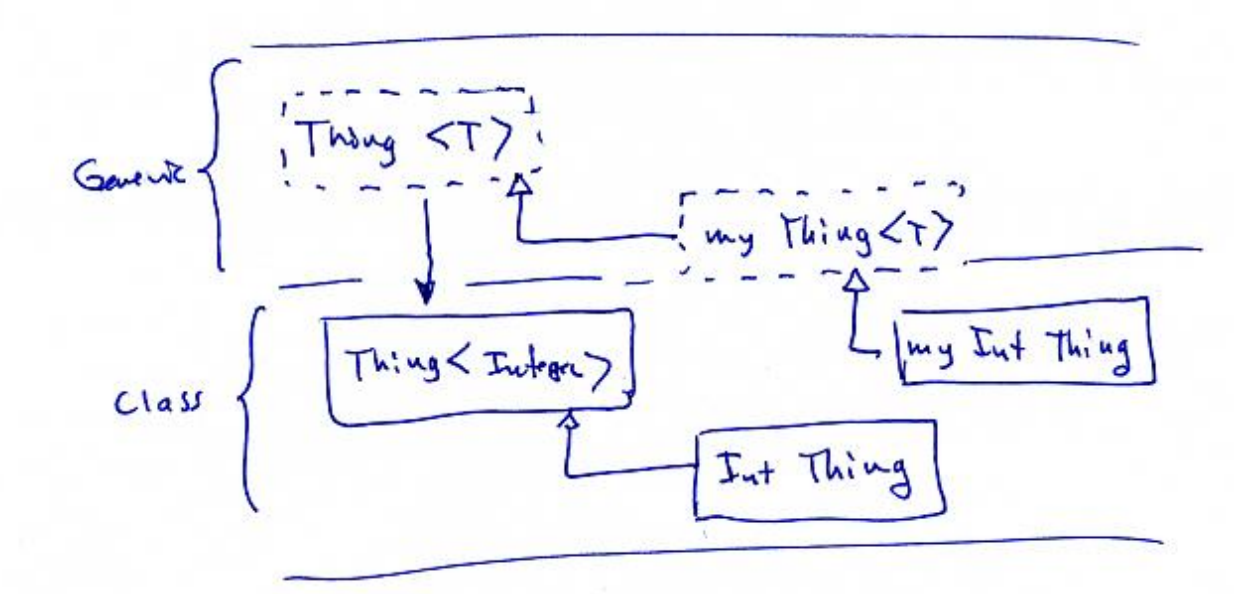
- Like all abstraction names, the actual name is unimportant (example of function parameters)



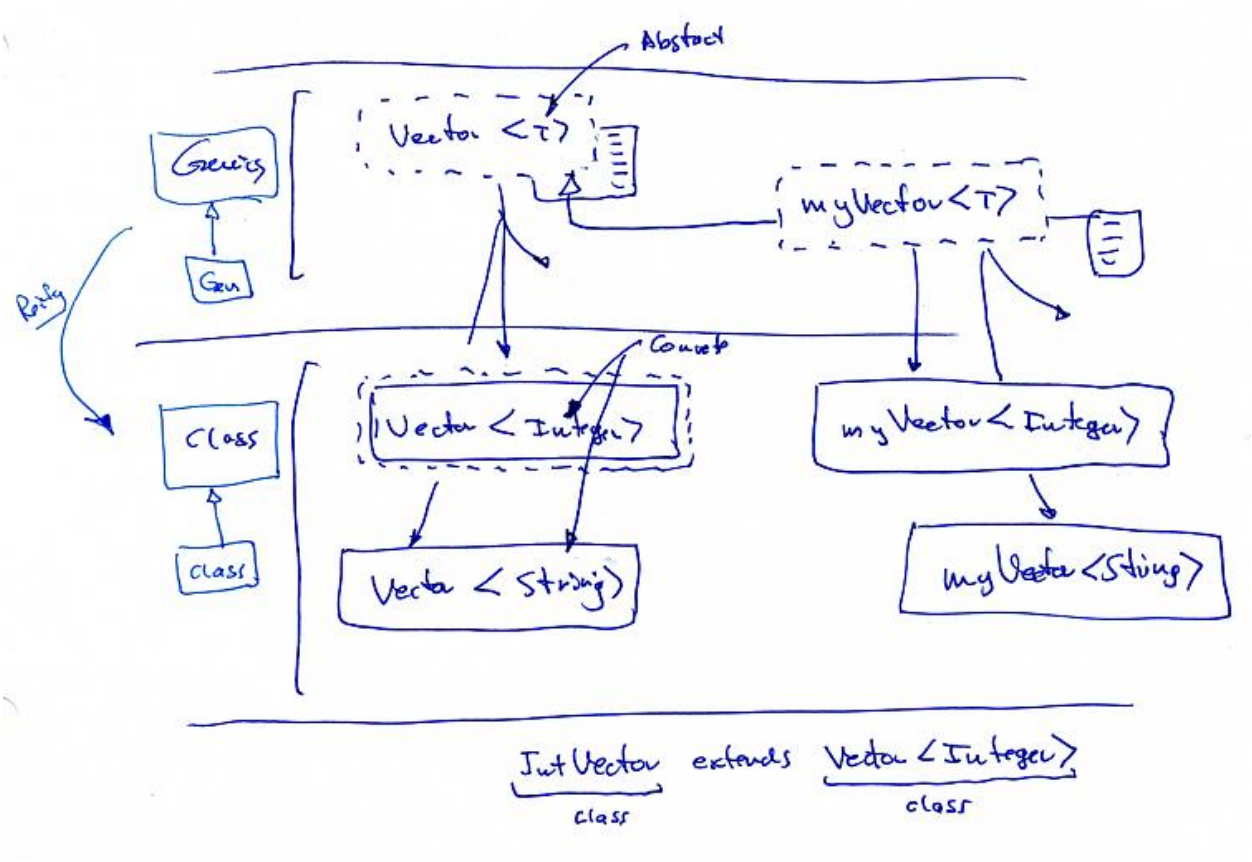
- Don't confuse type abstraction (parameter) names with concrete type names, the compiler won't! ☺



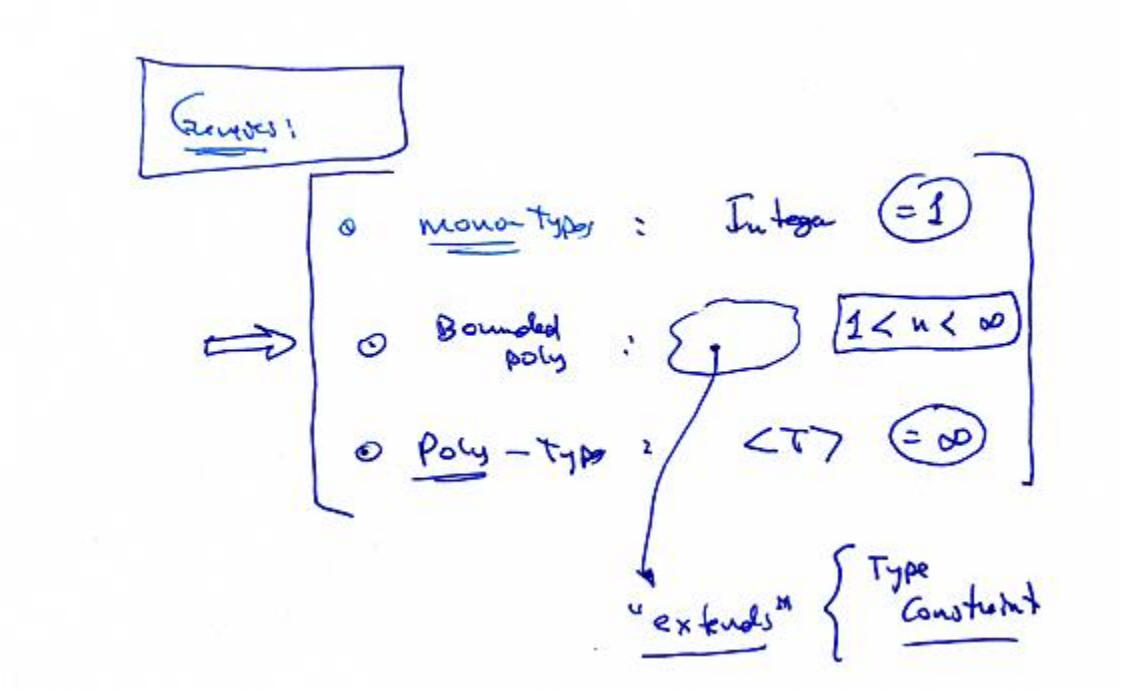
- Generics and Classes must be composed at their own levels



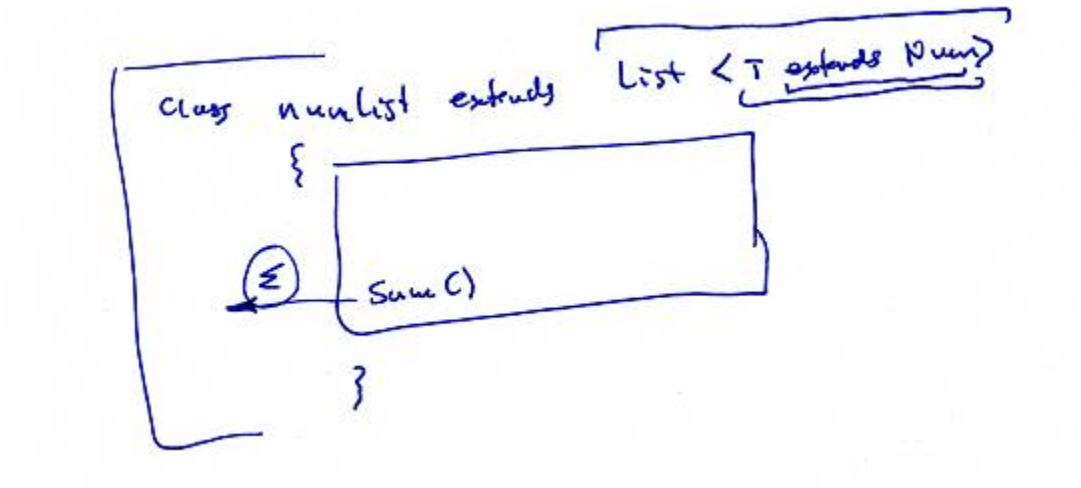
- Proper usage – instantiate a generic to a concrete, and then inherit


Abstract
Vector <T>
myVector<T>
Generics
Gen
Reify
Concrete
Class
Class
Vector <Integer>
myVector<Integer>
Vector <String>
myVector<String>
JustVector extends Vector<Integer>
class
class

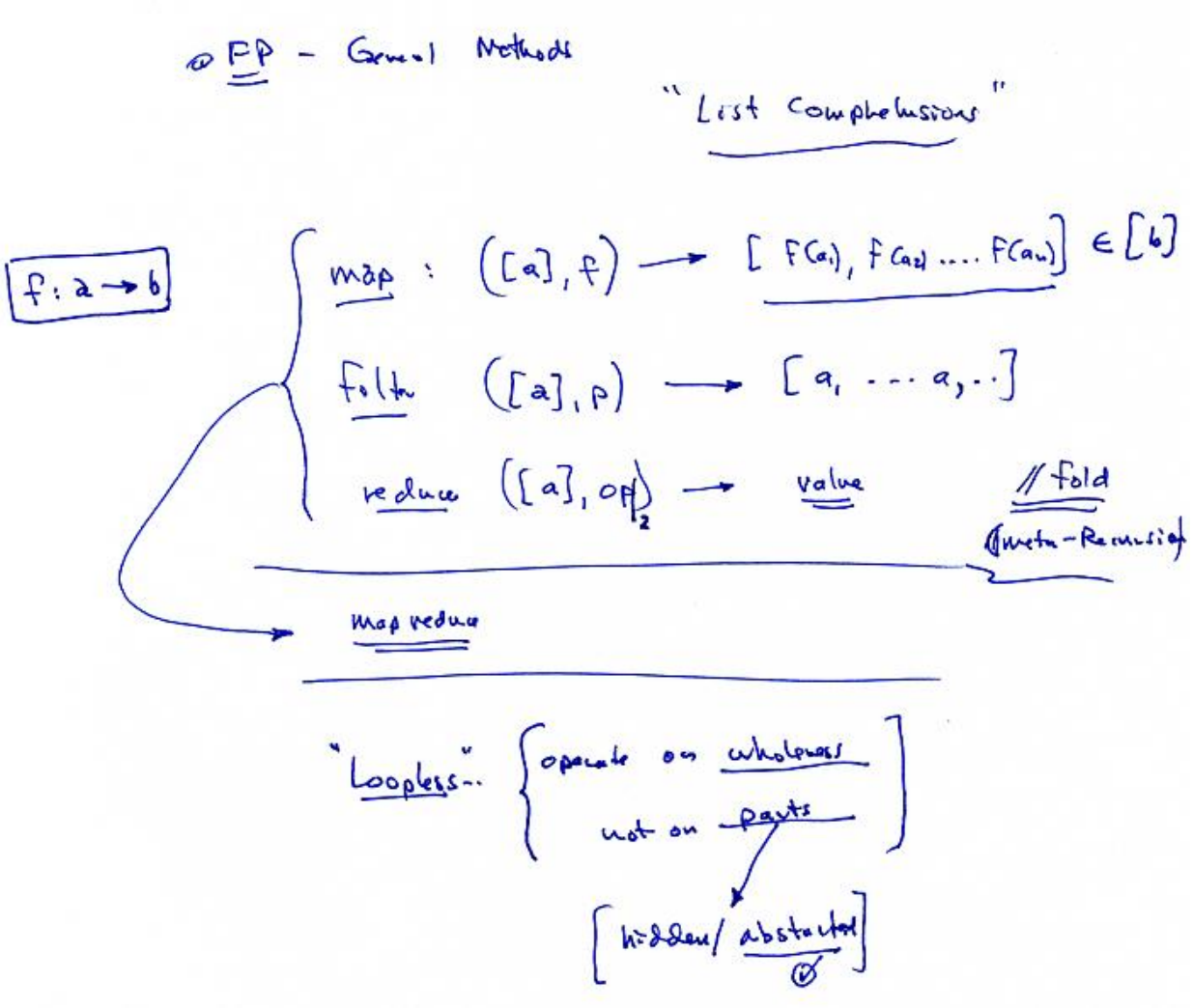
- Examples



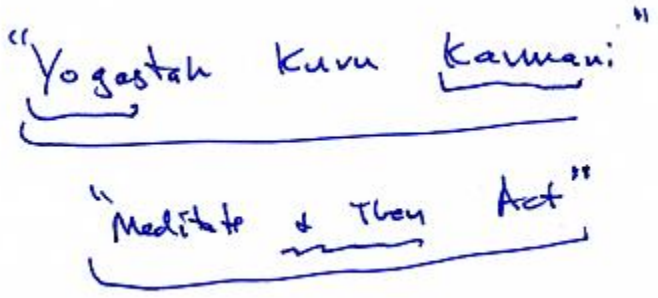
- Bounded generics



- Bounded Generics allow methods to be type specific



- Functional programming basics



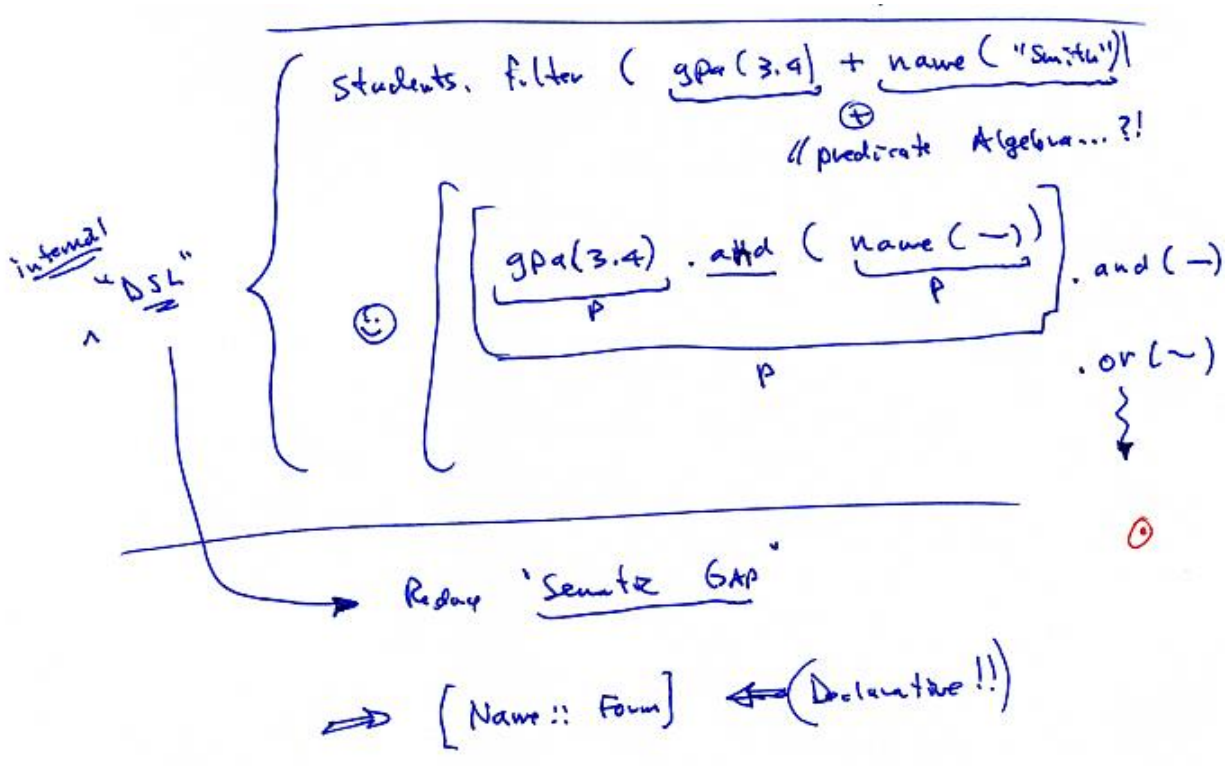
- Parallel SCI principle

Review of Iterator Exercise::

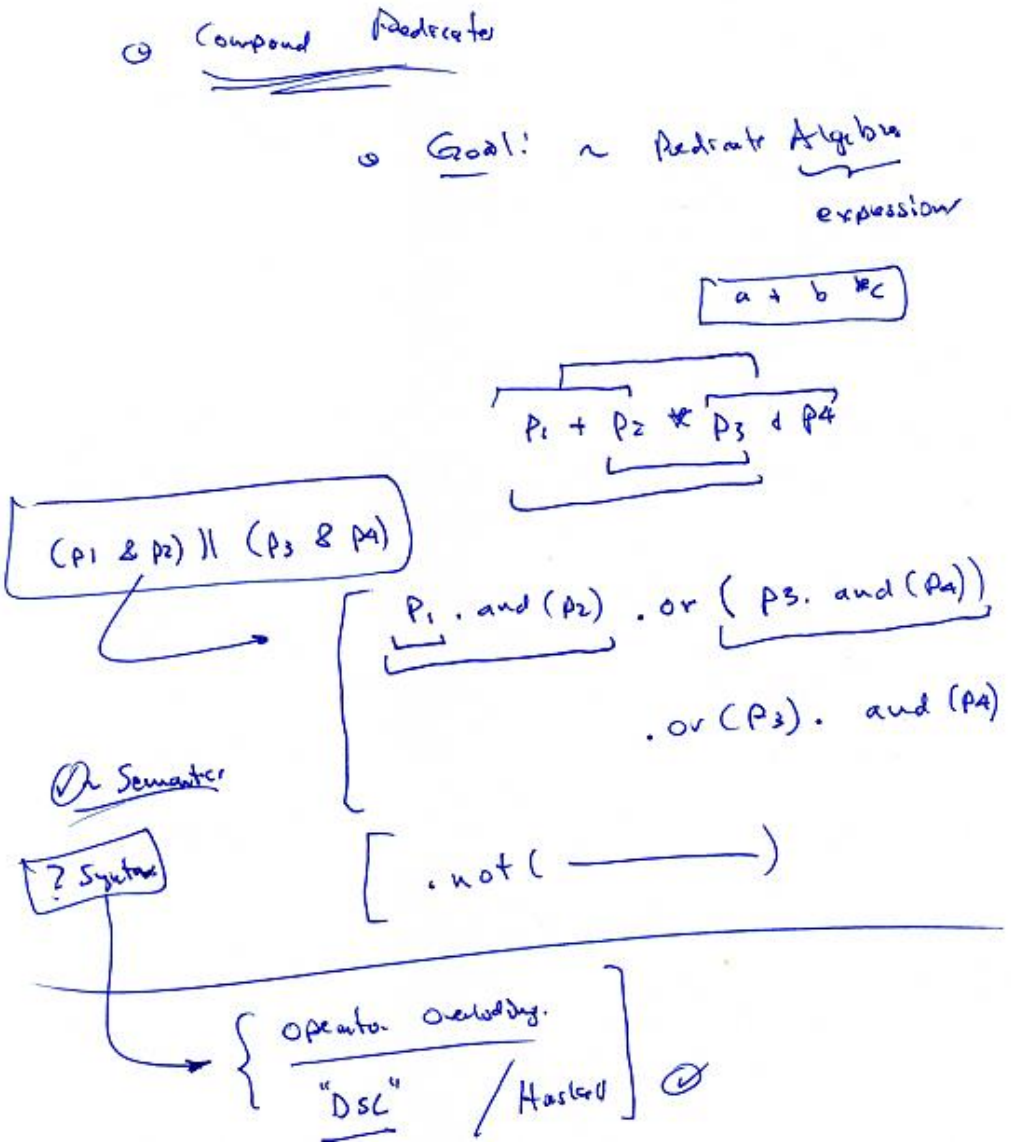
cs525: Pattern Notes

Iterator Styles:

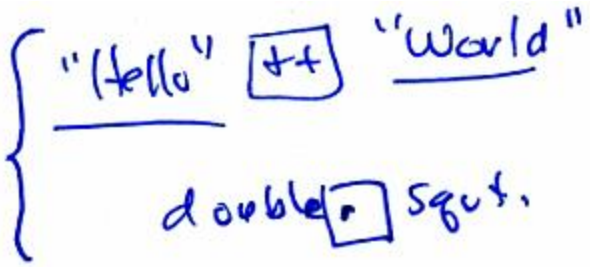
	Complete Traversal	Selective Traversal
External	Iterator<T> it = c.iterator(); while (it.hasNext()) ProcFun(it.next()) Or, *foreach* (T e : c) { … }	Iterator<T> it = c.iterator(Pred p); [Then use *normally*! ??]
Internal	c.doAll(ProcFun)	c.doAll(Predicate, ProcFun)



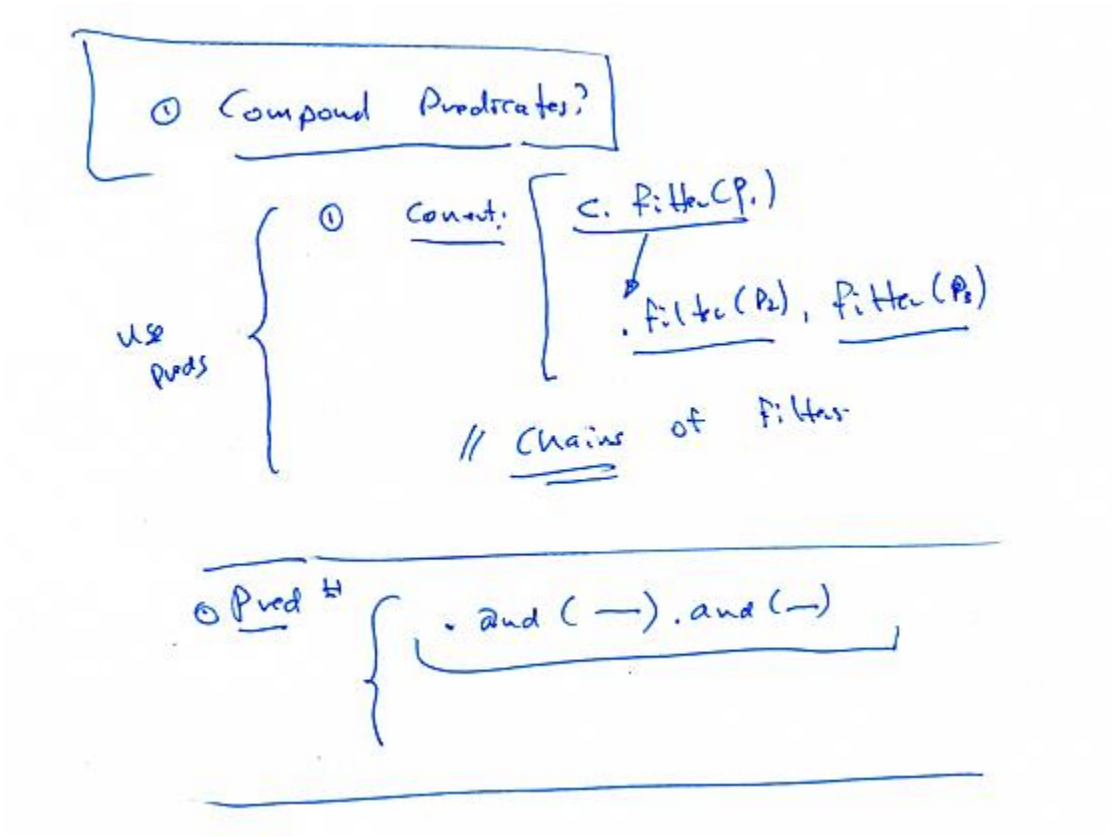
- Chaining of predicates



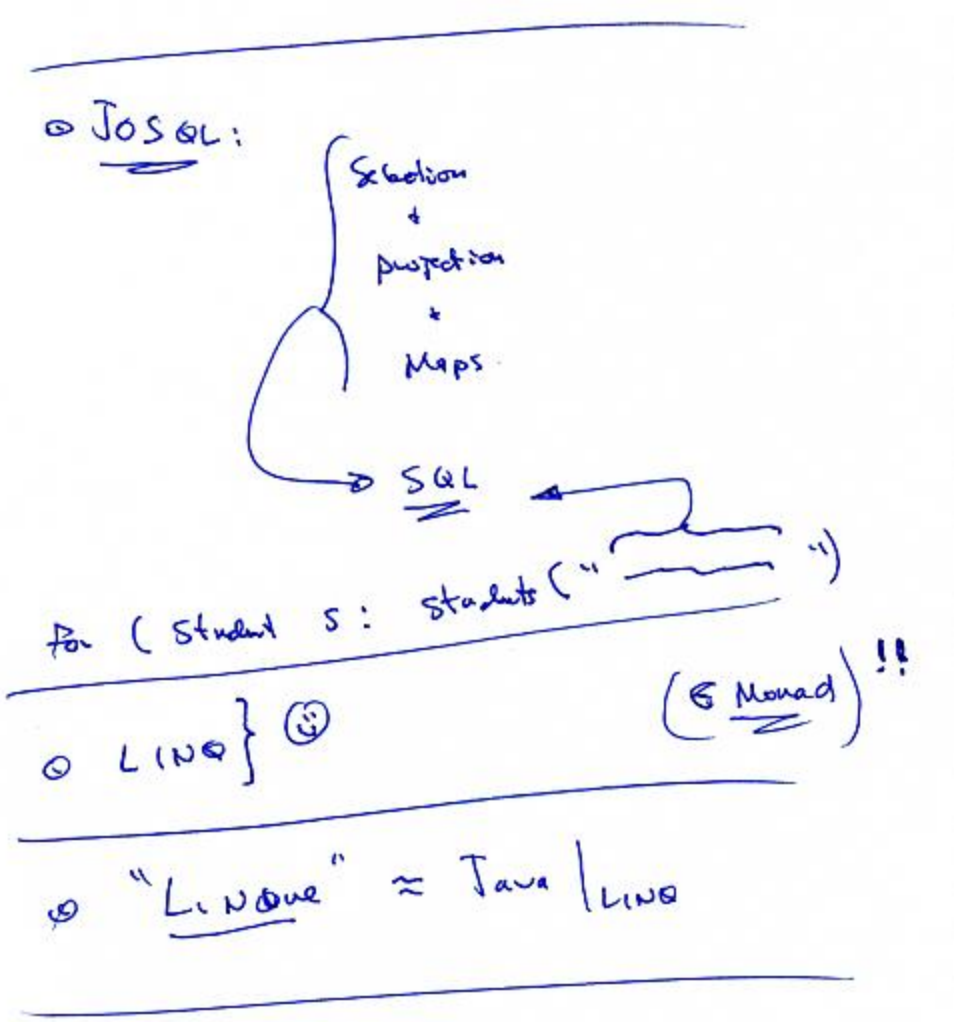
- Compound Predicates



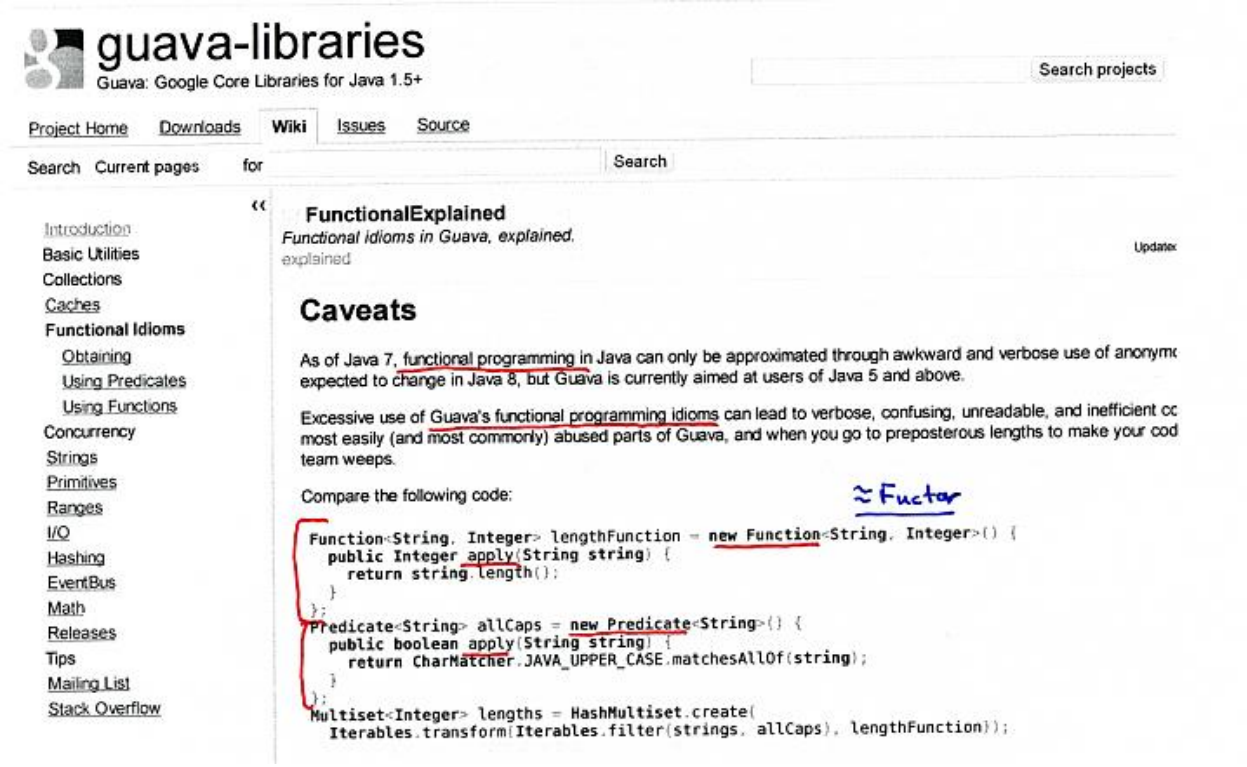
- We know how to compose data structures, but no convenient way to do similar things (in Java) with functions.



- How to add compound predicates



- Extending the idea of predicates to general queries



- 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_obje 1.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> f) {
        f.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) {
        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);
        }
}
```

- 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
⇨ functional arguments ≈ FP

- Modern languages like Ruby already have all of this…

- Summary of Iterator Exercise

Functional Programming:

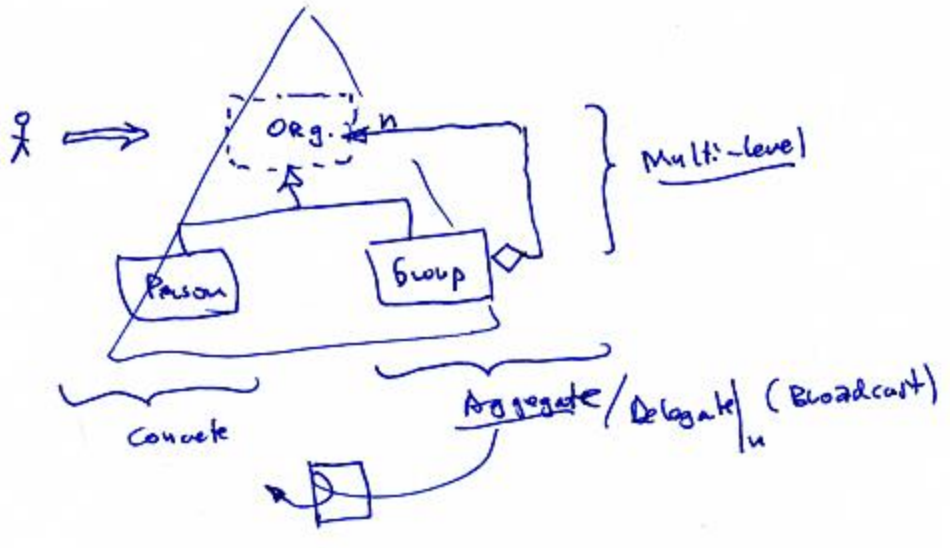
- An alternate programming paradigm
  - Dealing with wholeness, not parts
  - High level specification
    ⇨ not *over-constrained*
    ⇨ *parallelism, simplicity*
- Example: Internal & selective iterations
  - Are based on functional abstractions (functors, predicates)
  - Move in the direction of functional programming
  - *doAll(functor<T, R>)*
- Three standard functional methods
  - filter: *c.filter(predicate)* ⇨ c'
  - map: *c.map(func)* ⇨ c'
  - reduce: *c.reduce(f_2 , u)* ⇨ value
- All can be easily added to standard collections
- Reduce ≡ *fold { foldr, foldl }*
  - *Fold* ≈ 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

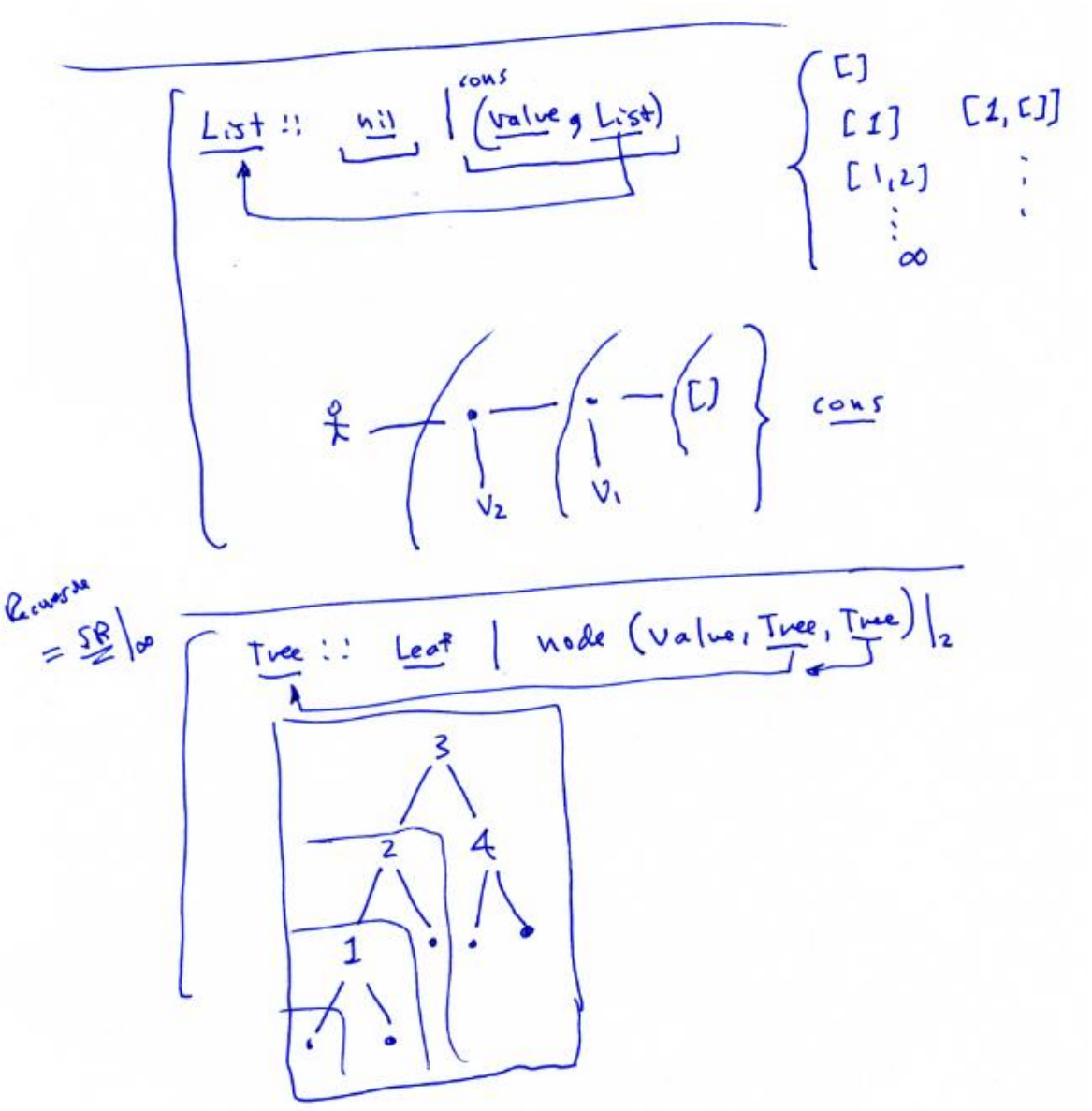
Daily Topics

Day 7b: Composite Pattern

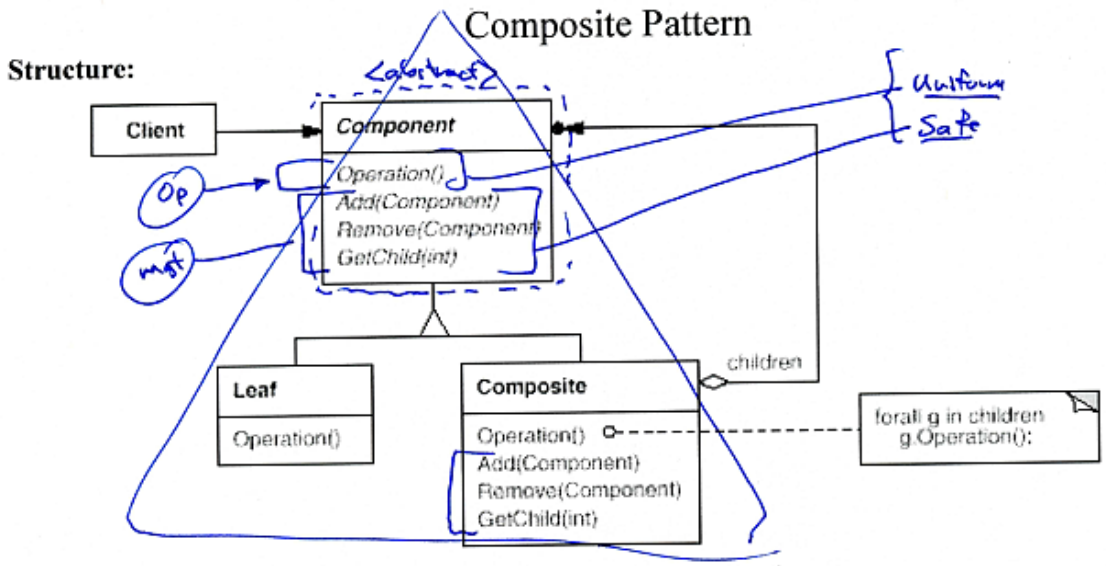
- Quiz & Review
- General representation of recursive data-structures
  - list, tree
- ⇨ Composite pattern
  - parts ≈ whole
  - based on SR loop ⇨ ∞ composition
- Design options
  - uniform, *safe* GoF
- Recursive iteration?
  - internal ?
  - external ?? ☹
- Lab:
  - Composite Lab



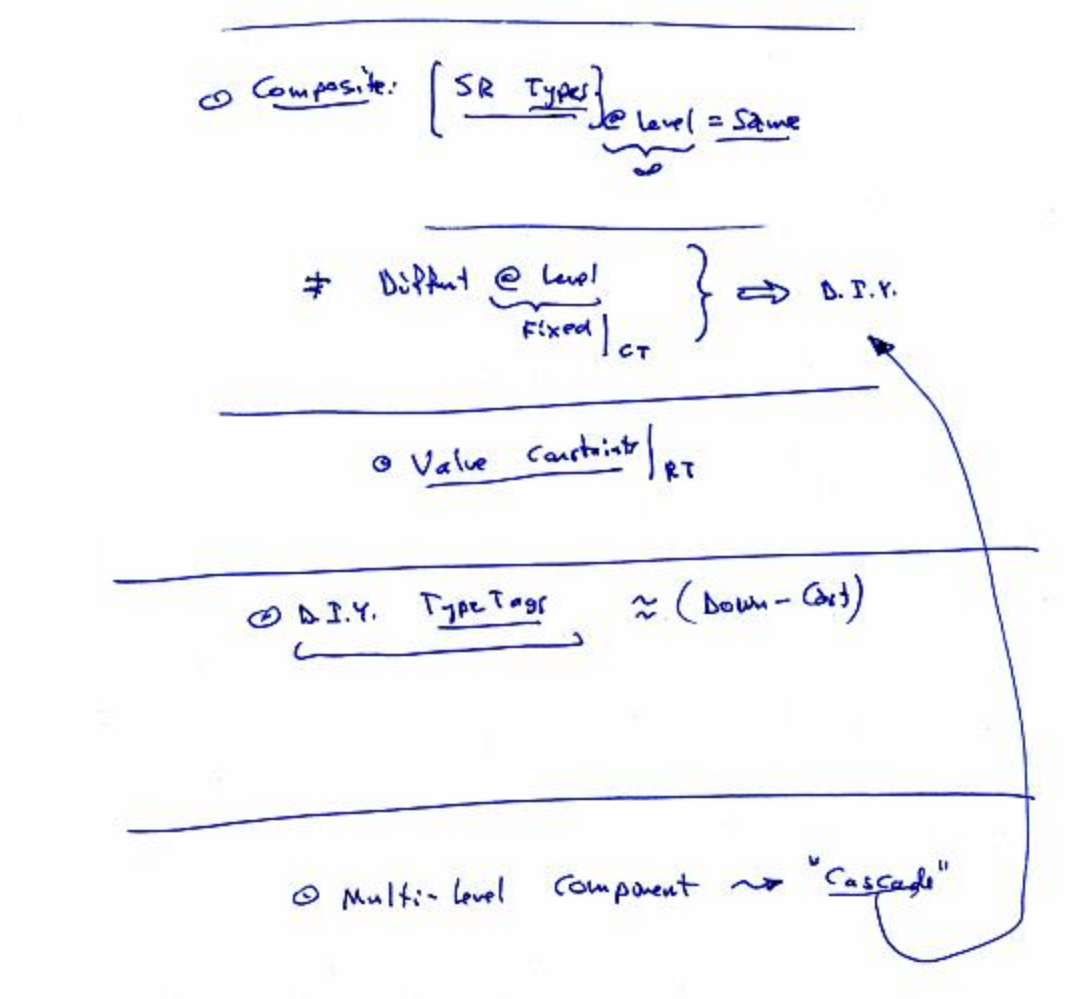
- General idea of Composite
- Recursive types, unified parts *& Whole



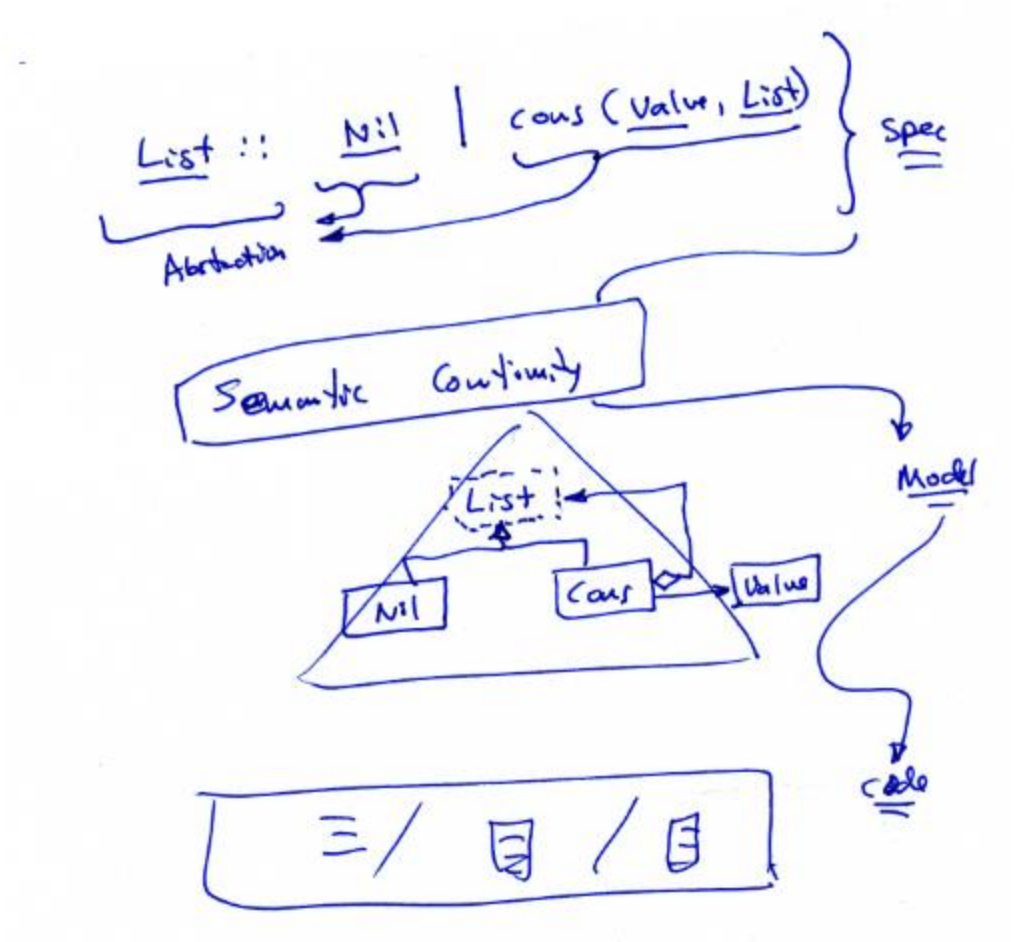
- Formal model of recursive types



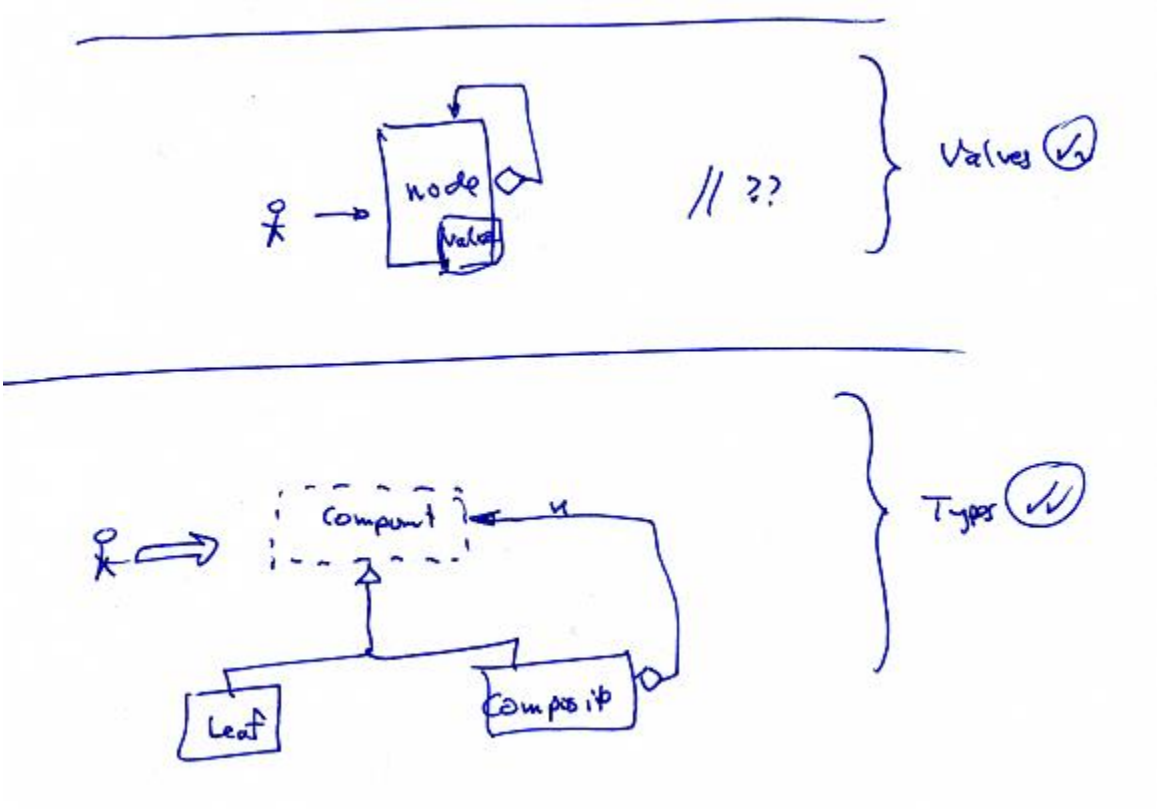
- GOF description of Composite Pattern



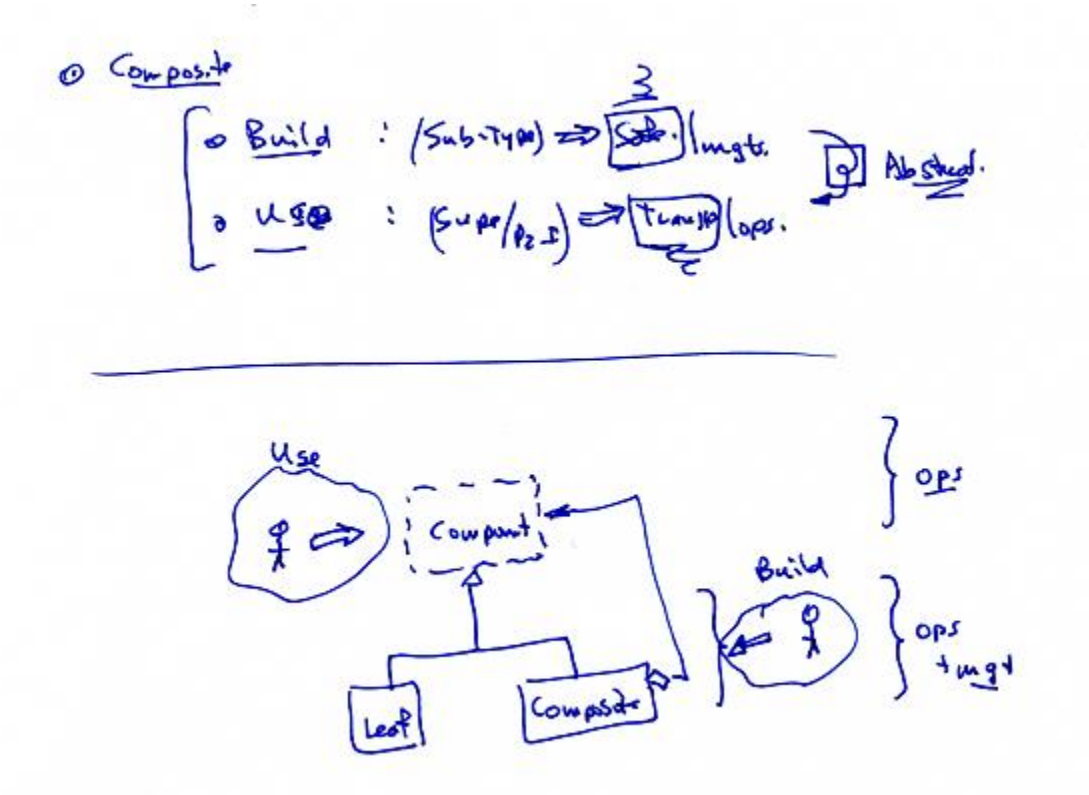
- Composite models systems with the same types at every level



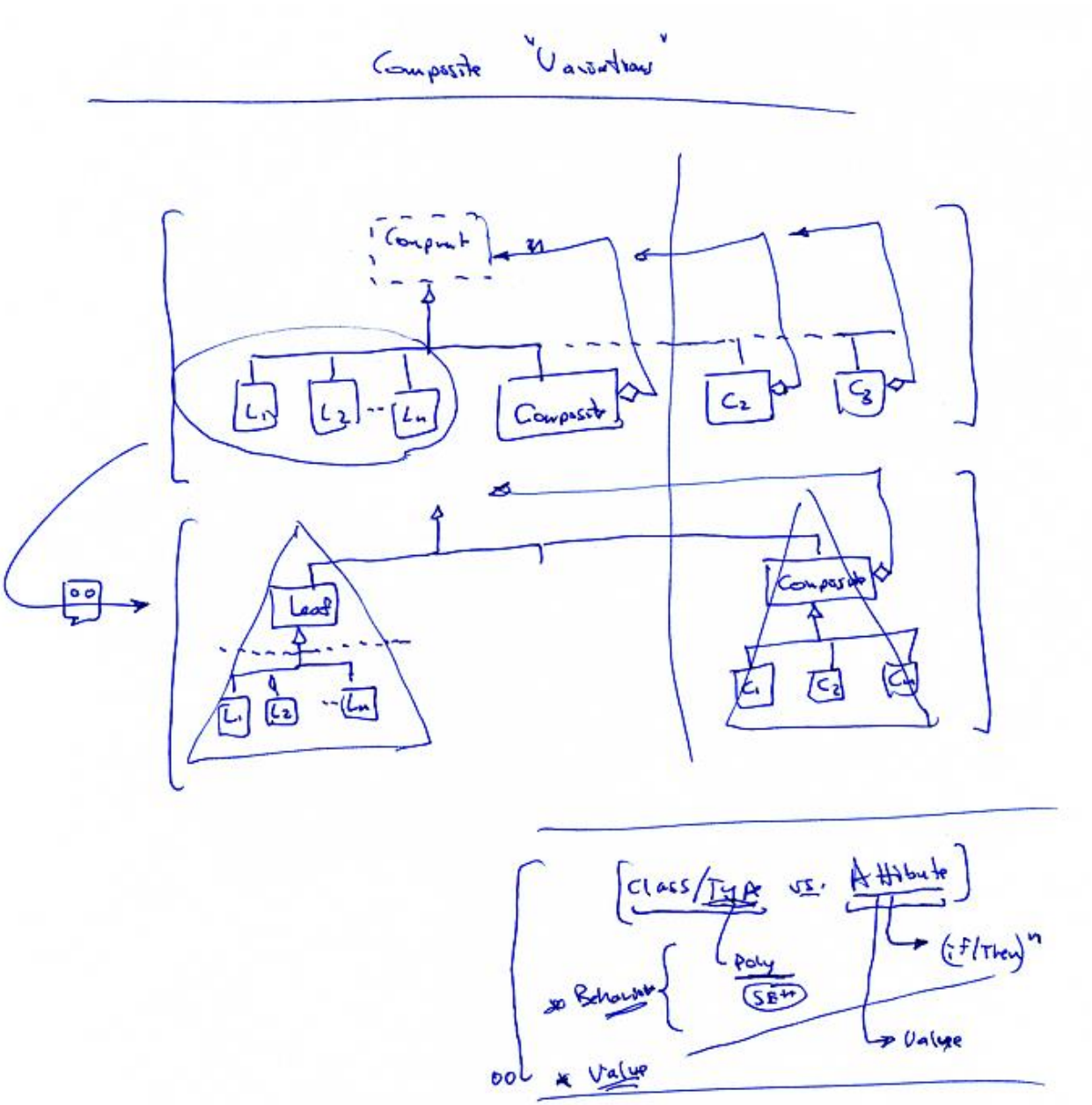
- Composite can be derived as an OO model of an abstract definition



- Composite models the business (application) domain



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

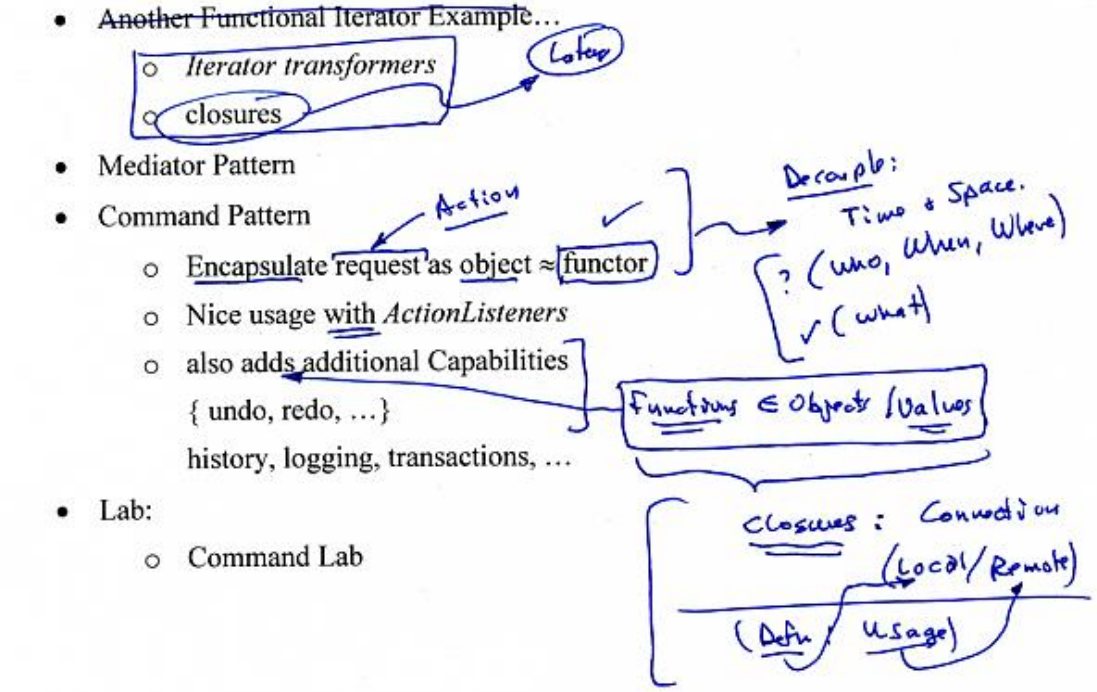


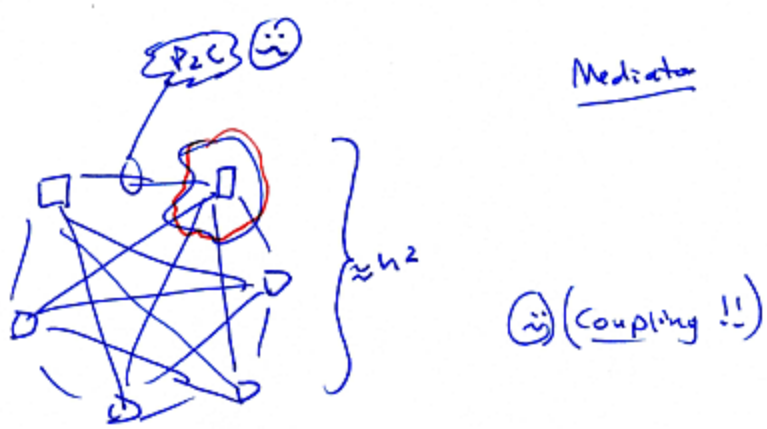
- Variations on Composite Pattern

Day 8: Command & Mediator Patterns

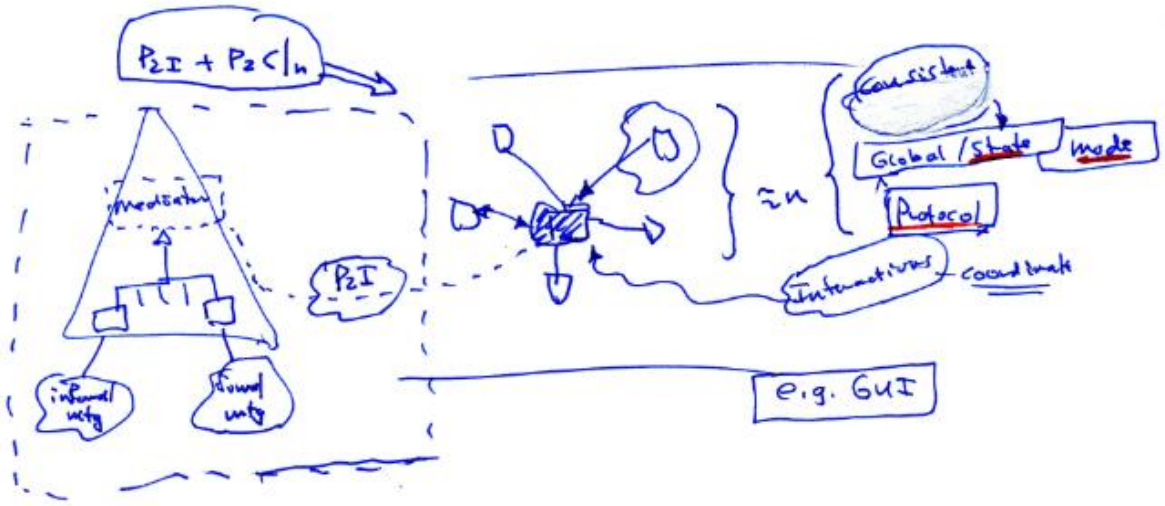
Day 8: Command & Mediator Patterns

- 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, …
- Lab:
  - Command Lab



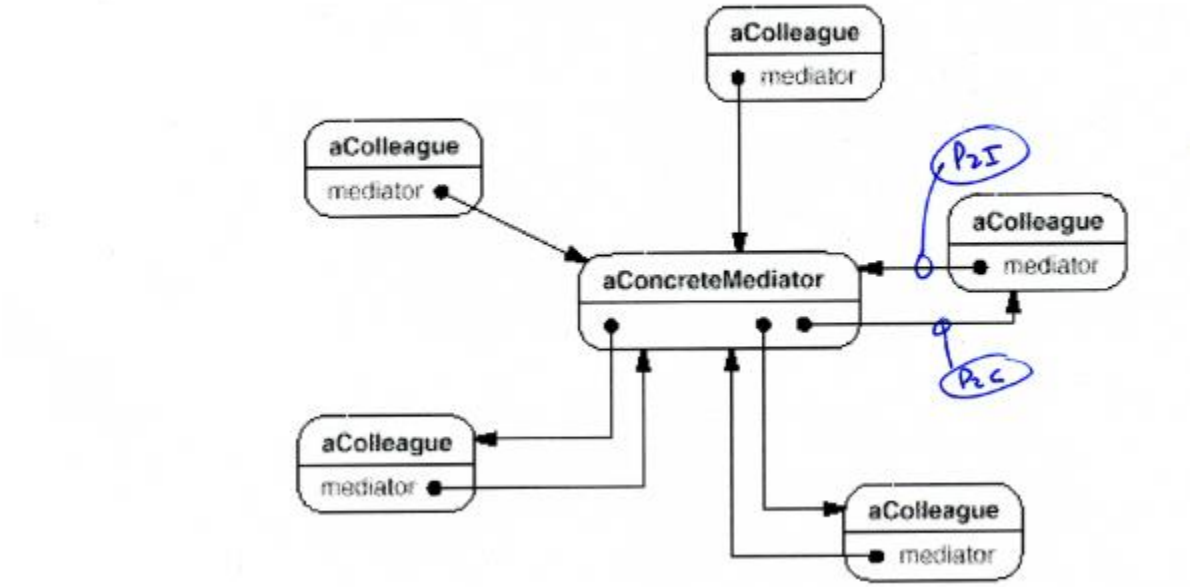


- Mediator helps resolve coupling in a sub-system

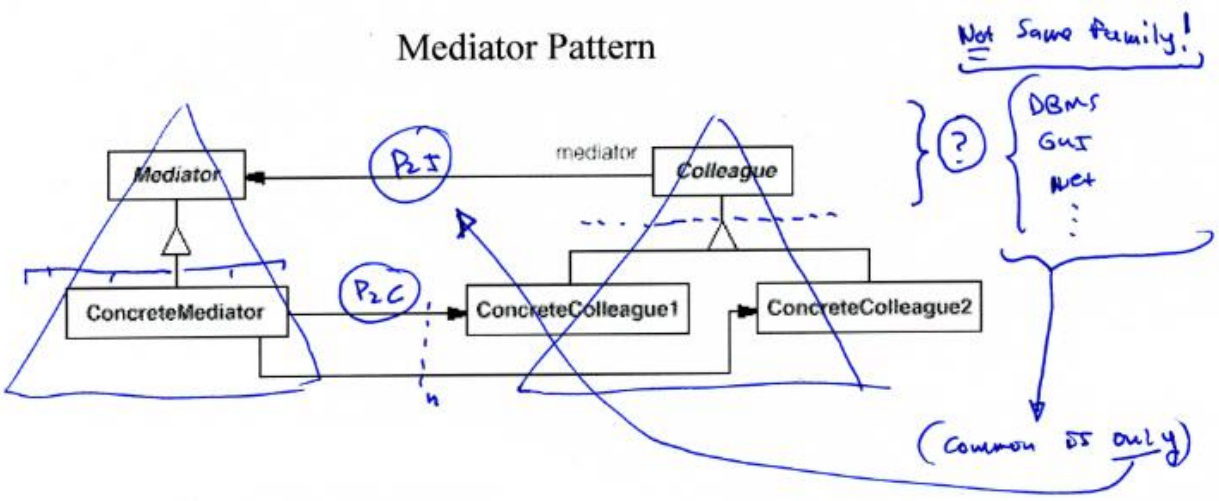


- Mediator centralizes protocol, coupling, and state (dynamically)

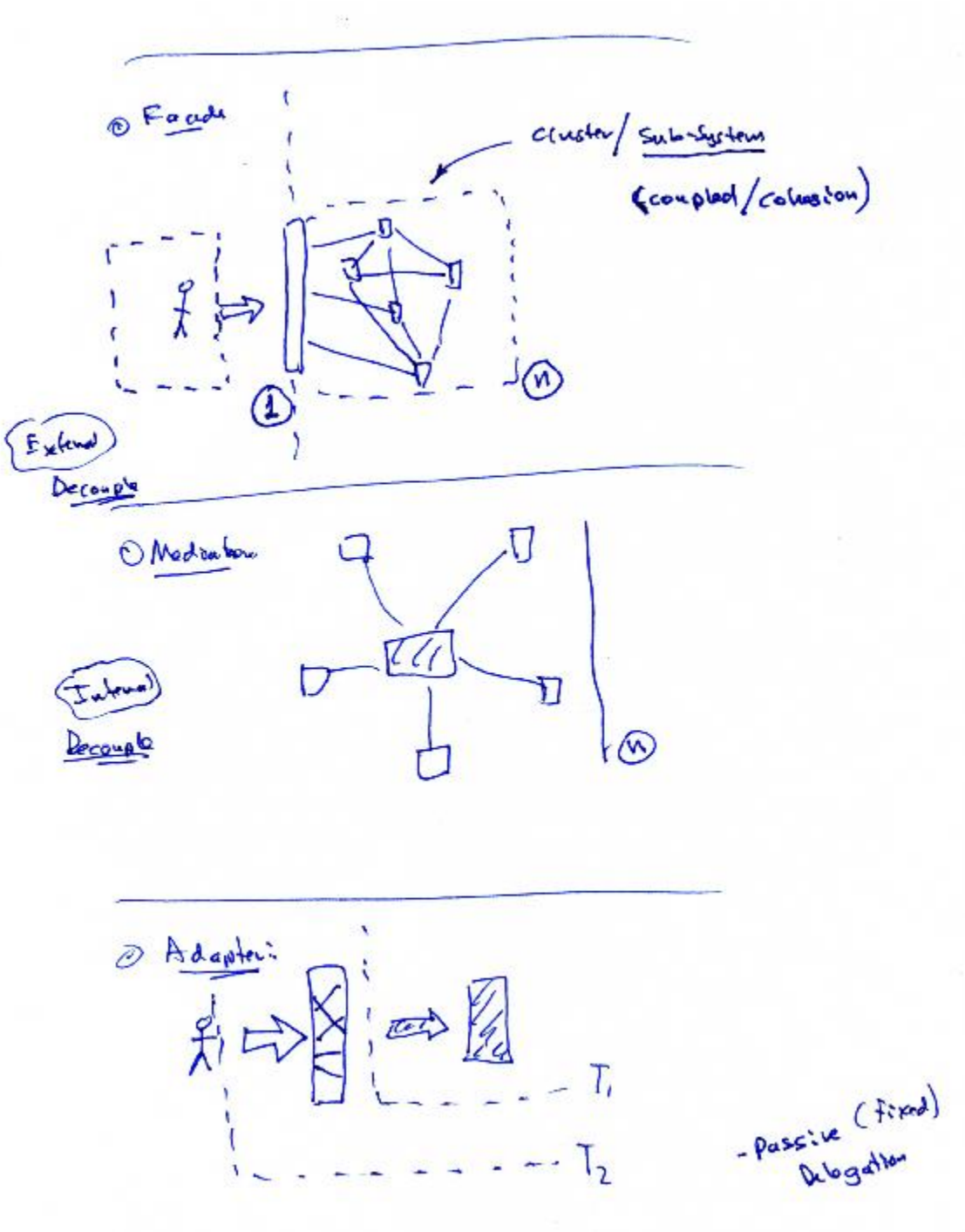
A typical object structure might look like this:



- GOF example

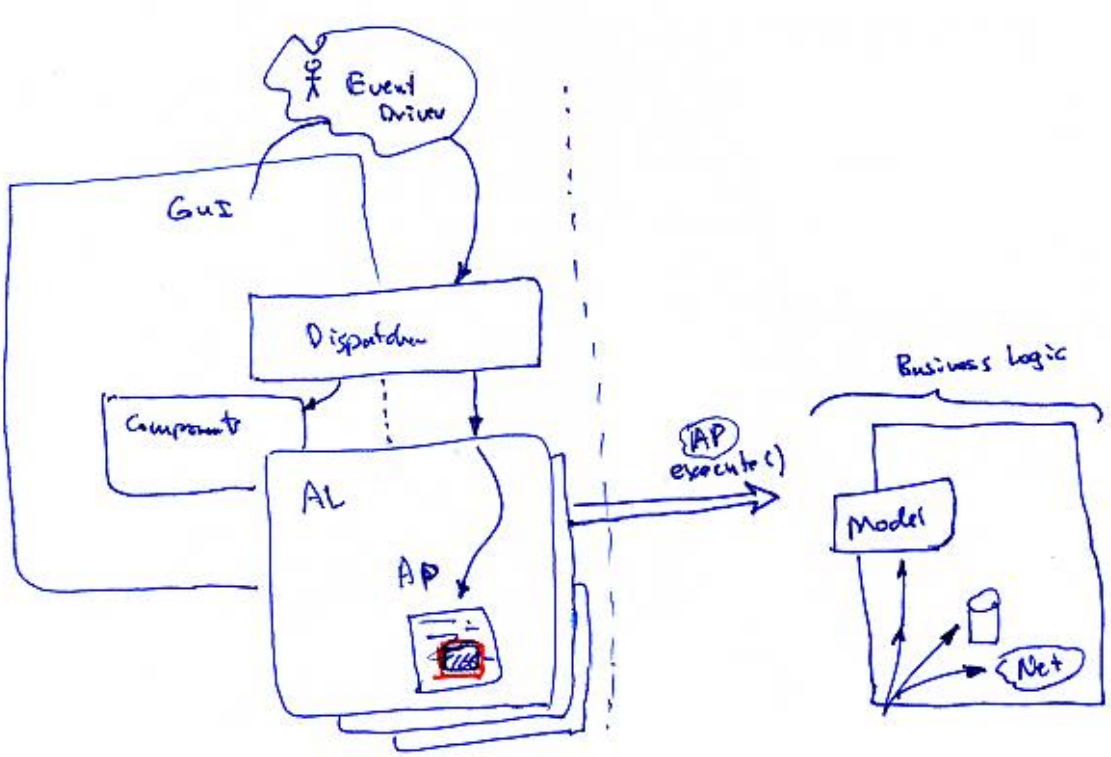


- GOF mediator

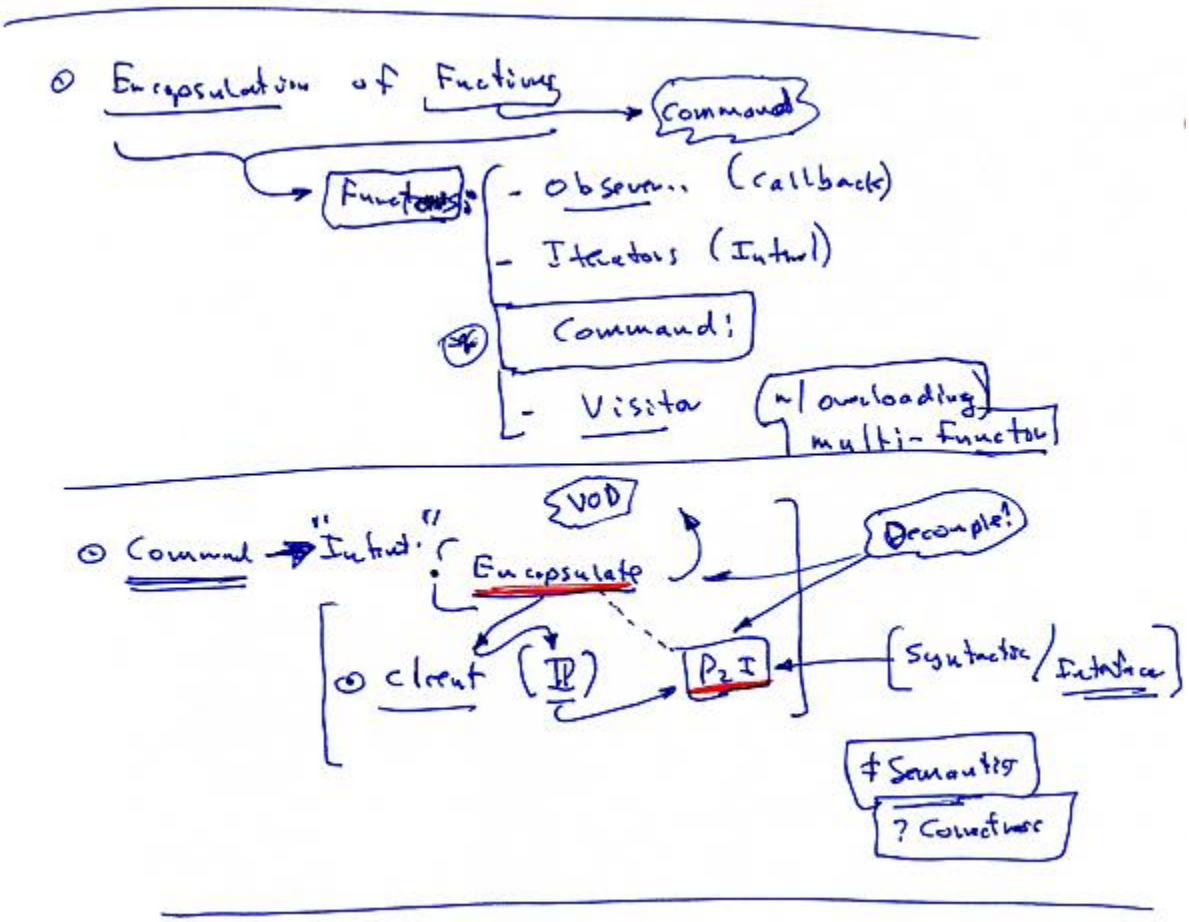


- Comparing Mediator and some other patterns

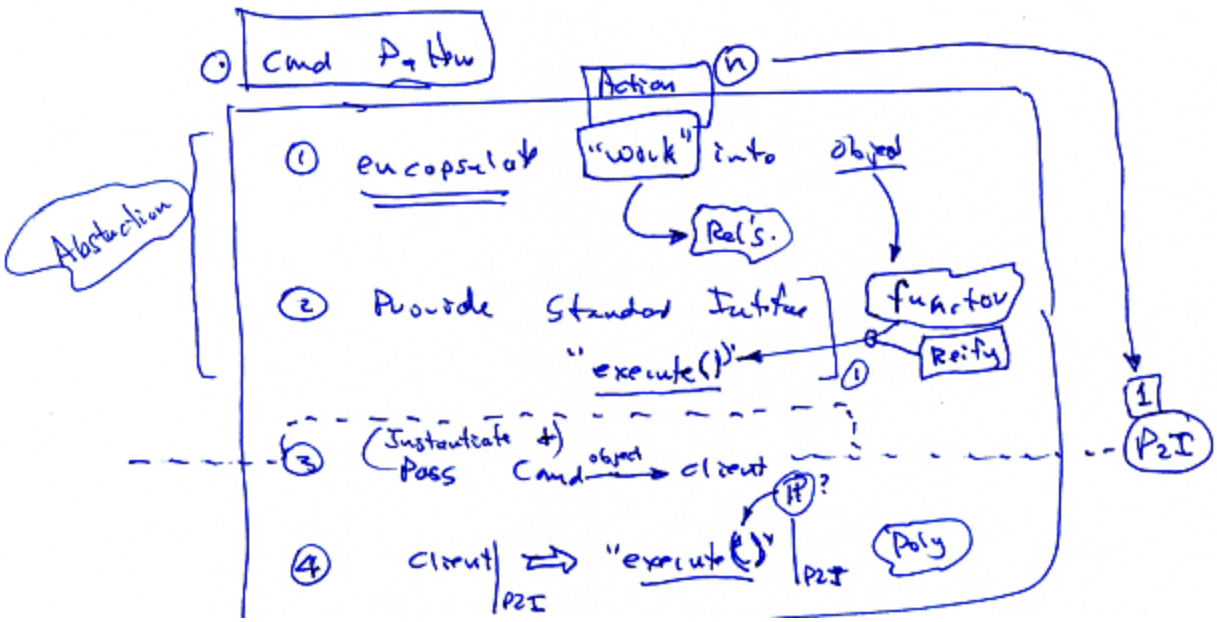
Command Pattern::



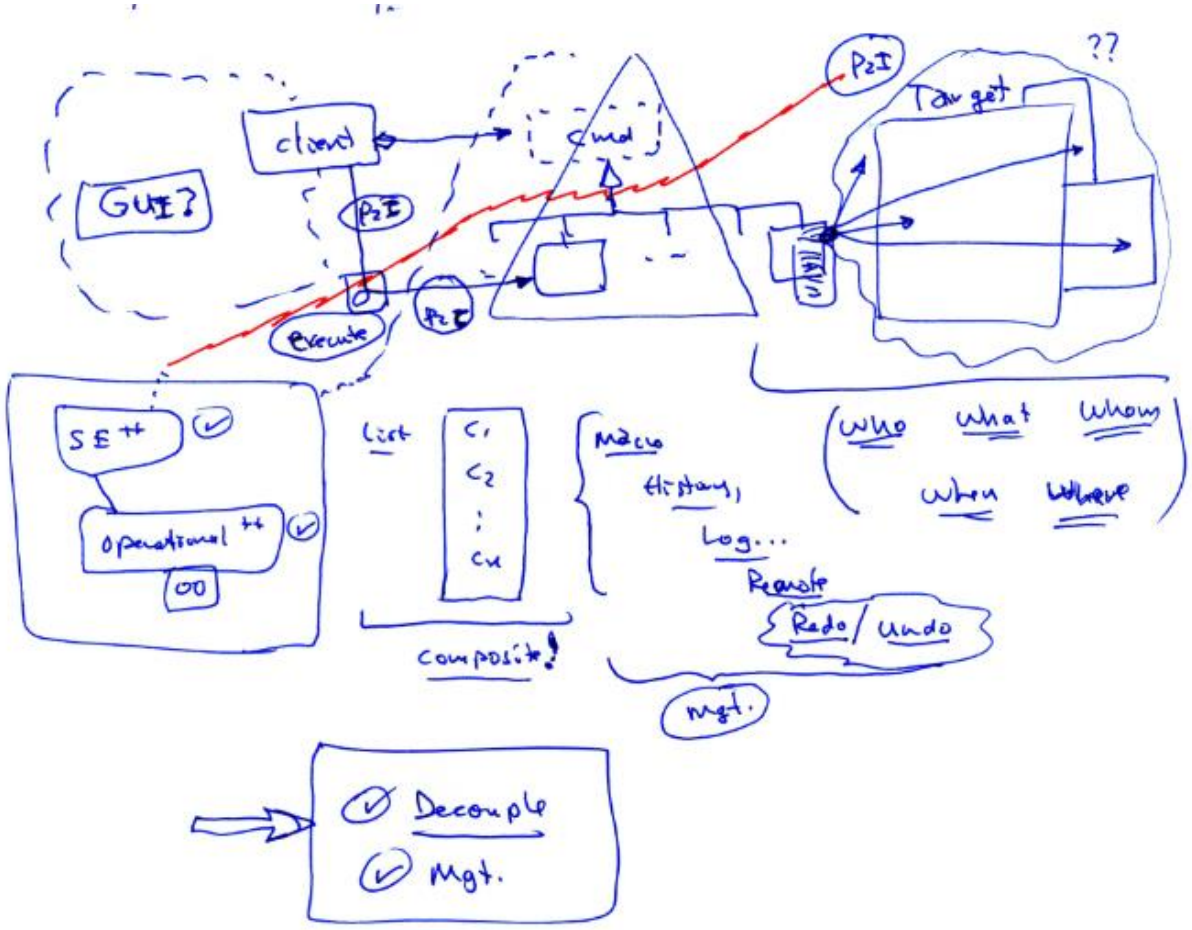
- The original Observer pattern
- AP ≈ Command?



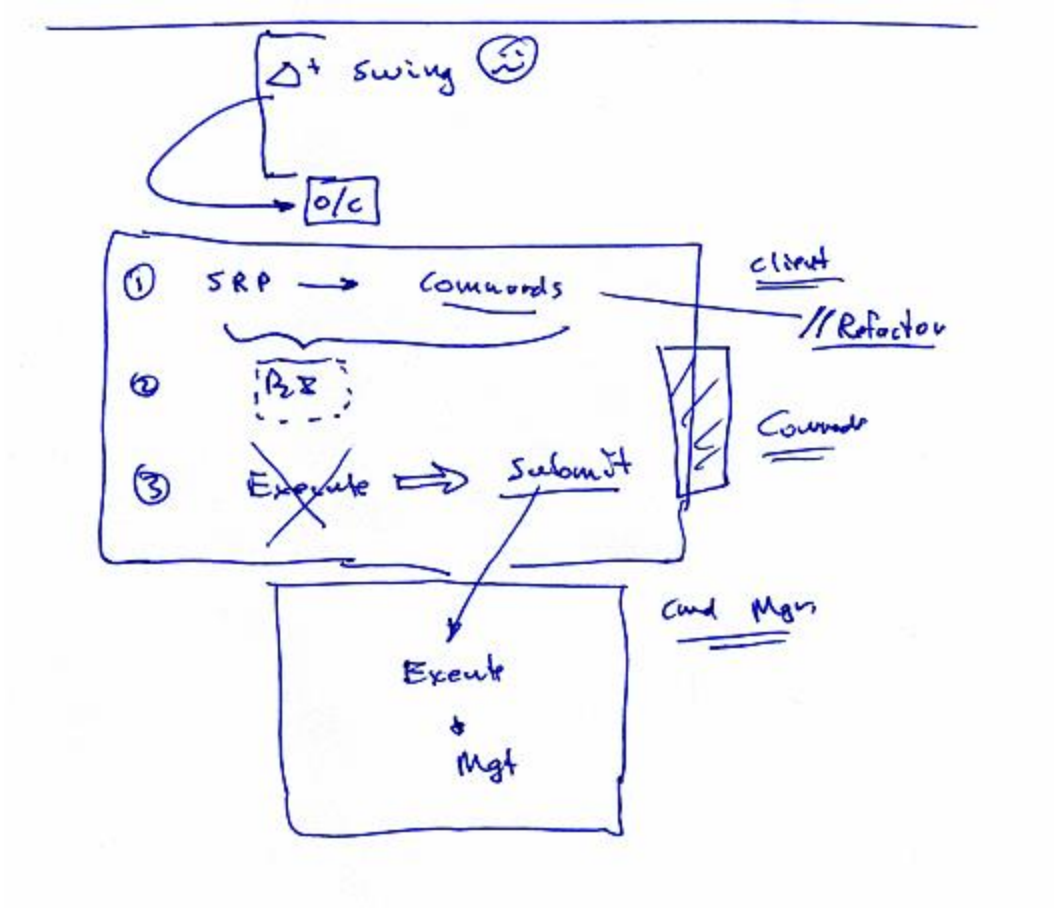
- Basic ideas of Command Pattern



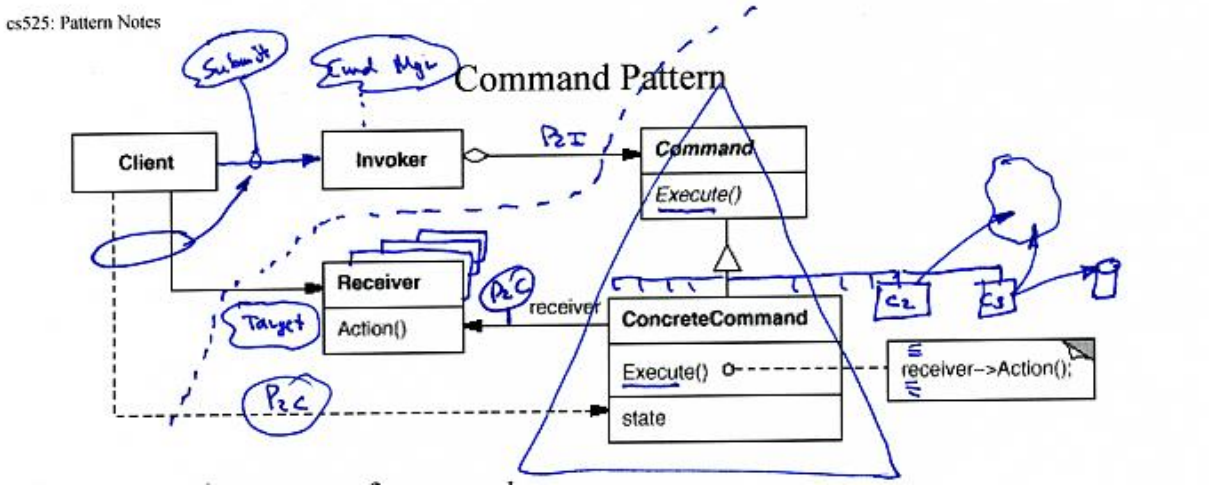
- Steps of usage of Command pattern



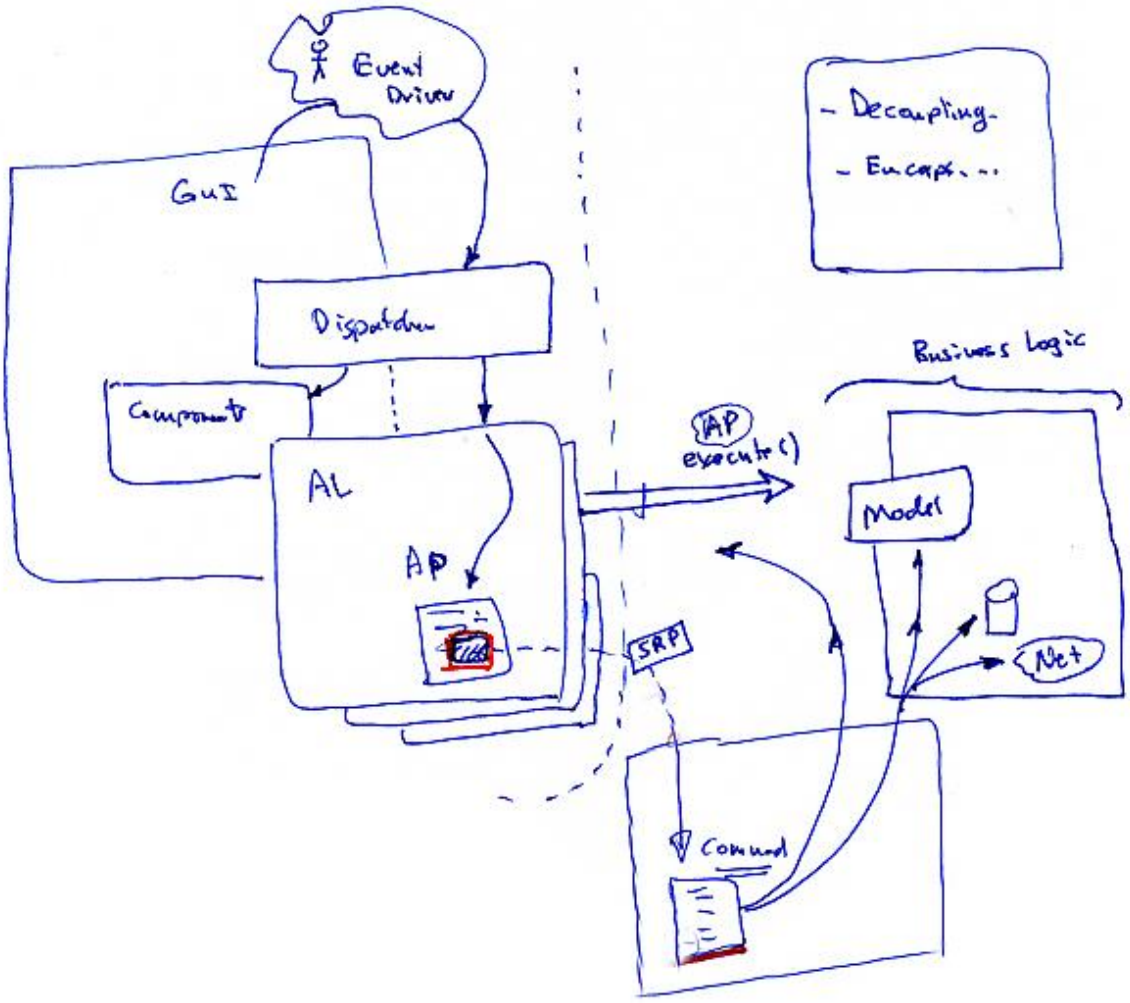
- Overall Architecture and usage of Command Pattern



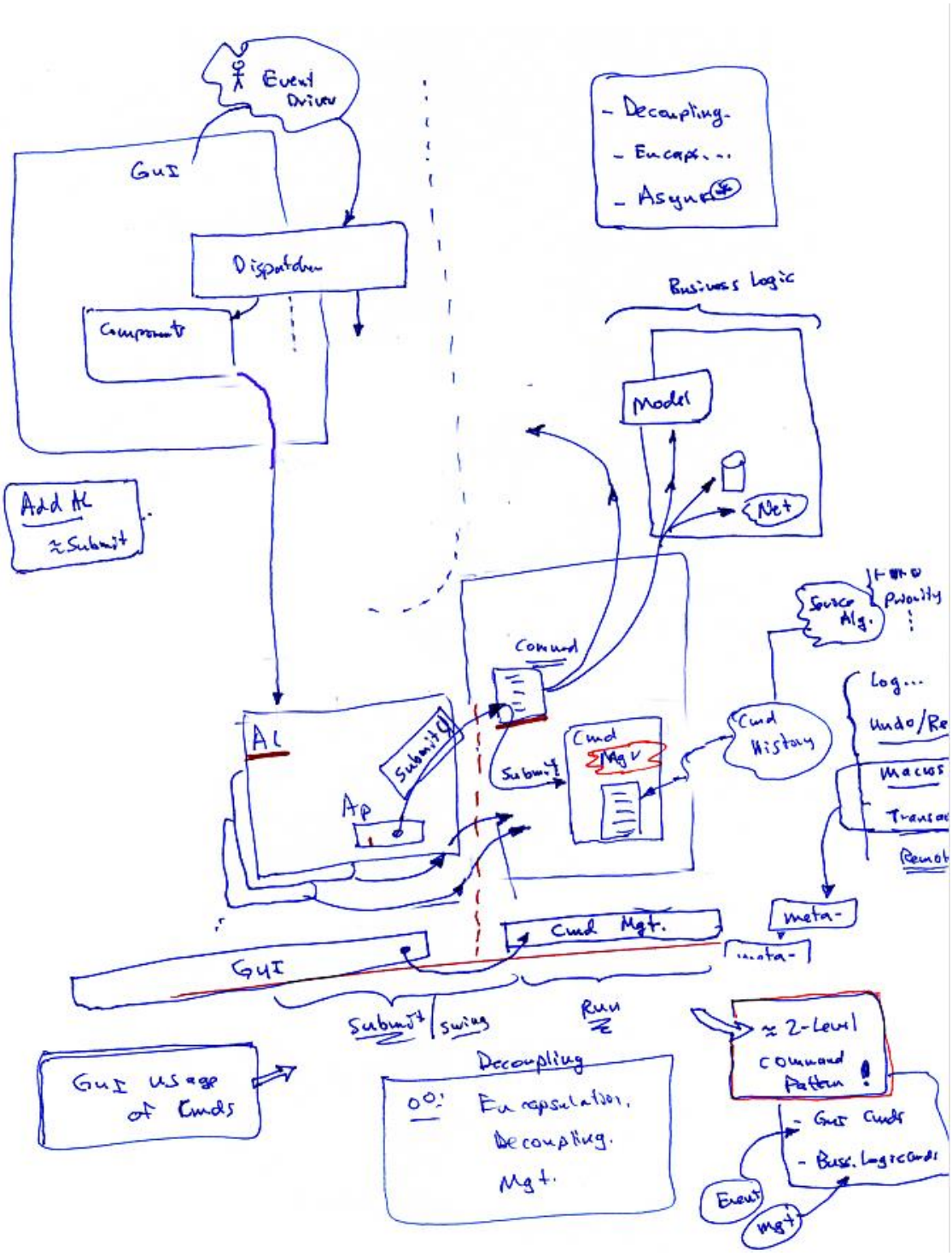
- Our lab uses GUI + Command pattern



- GOF structure for Command Pattern

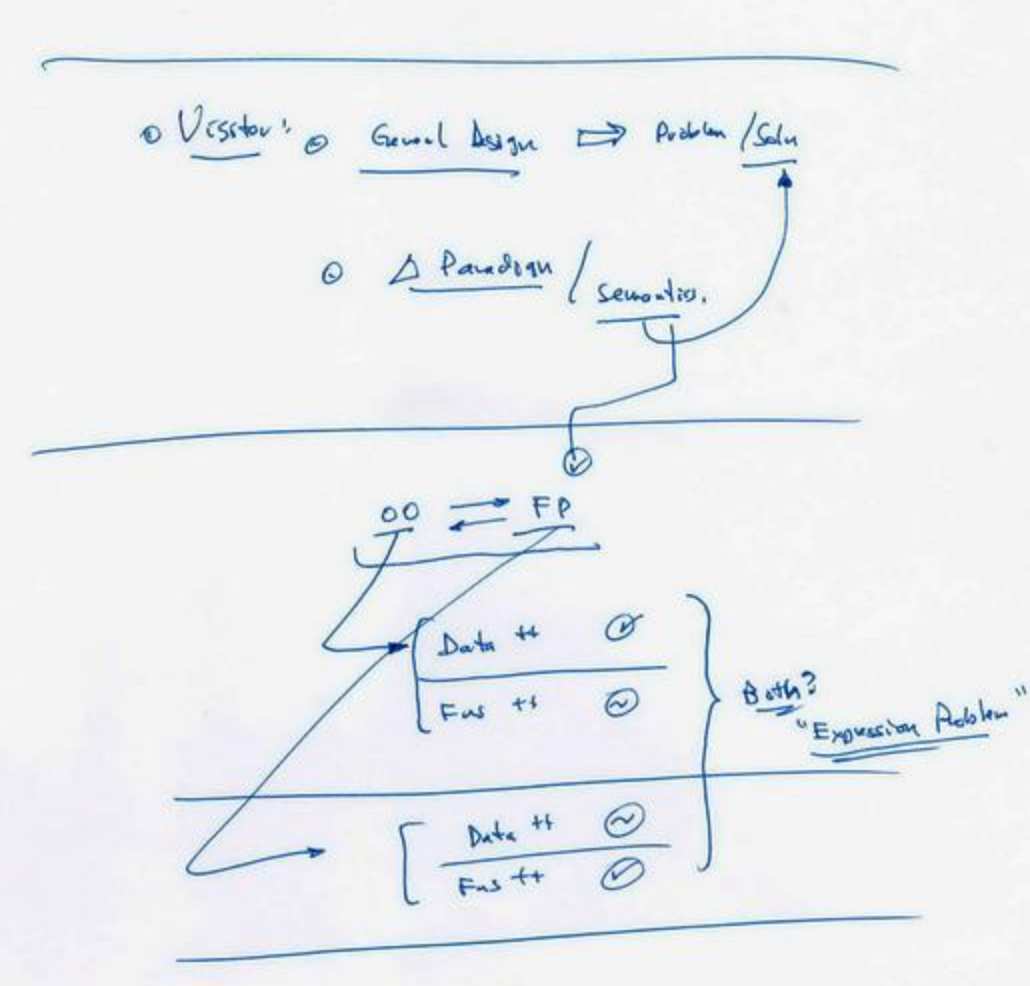


- Modify to separate AP from AL, Business logic from GUI

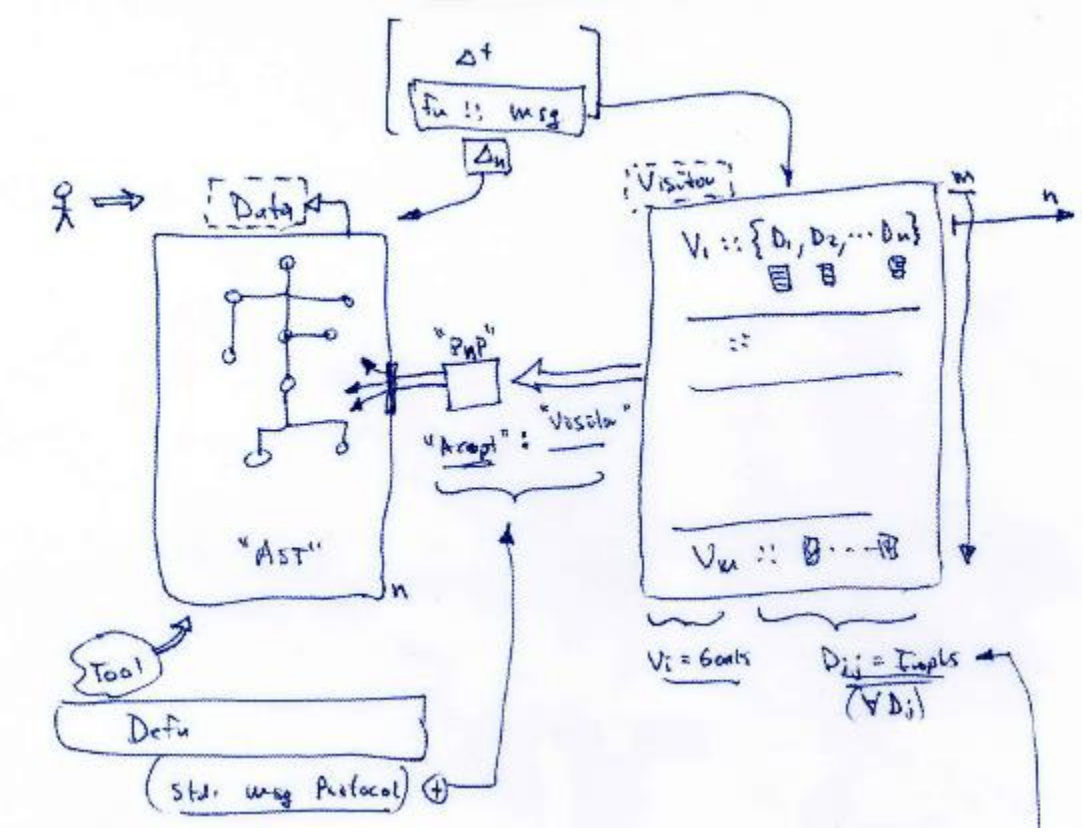


- Rewrite to use Command pattern
- Keep GUI dispatching using Observers

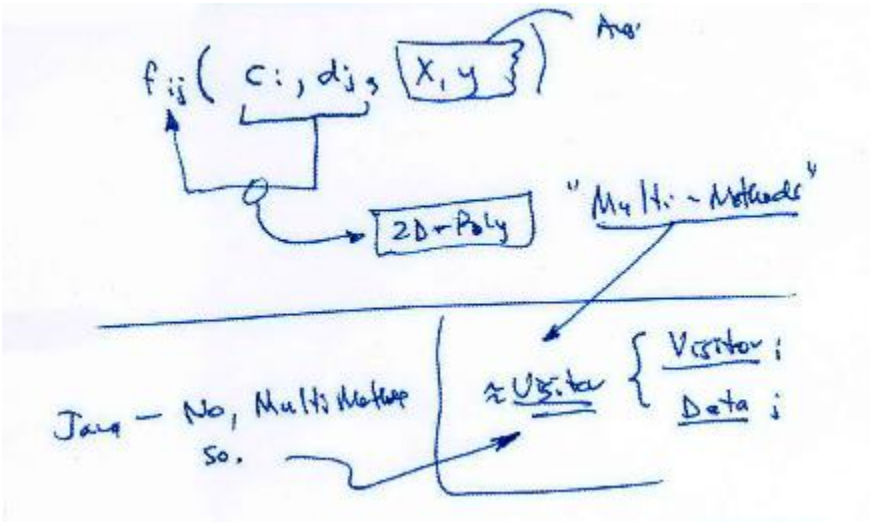
Visitor Pattern::



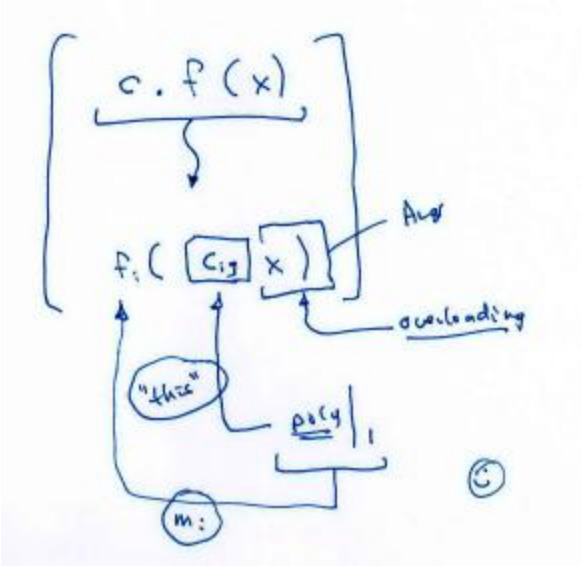
- Visitor Pattern; ≈ non-OO → Open/closed on functions instead of data



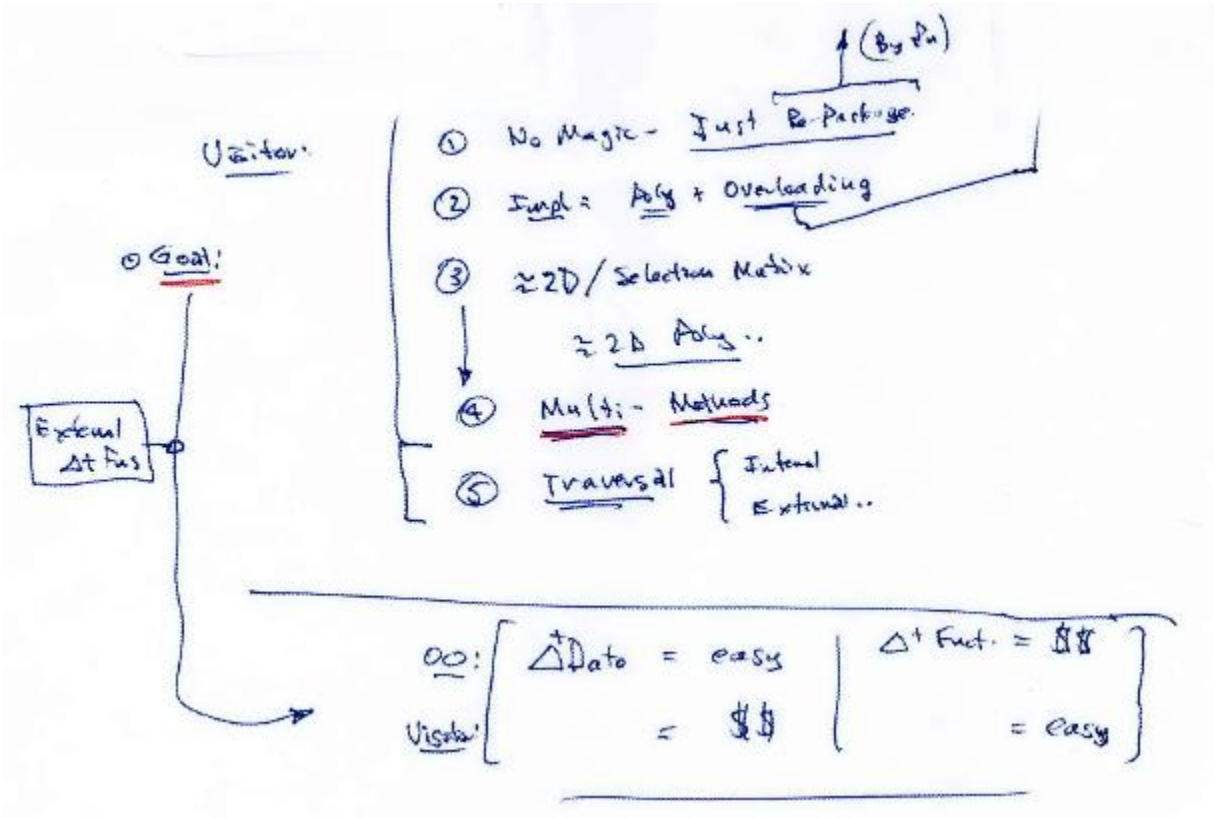
- Architecture of the Visitor Pattern



- One could consider poly dispatch on multiple arguments… ⇨ *multi-methods*



- Standard polymorphism; is single argument poly dispatch

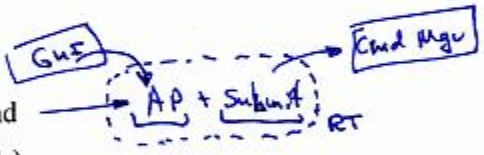


- Summary of results of Visitor Pattern

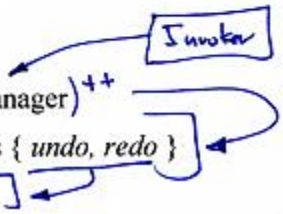
Day 9: More Command Pattern

Day 9: More: Command Pattern

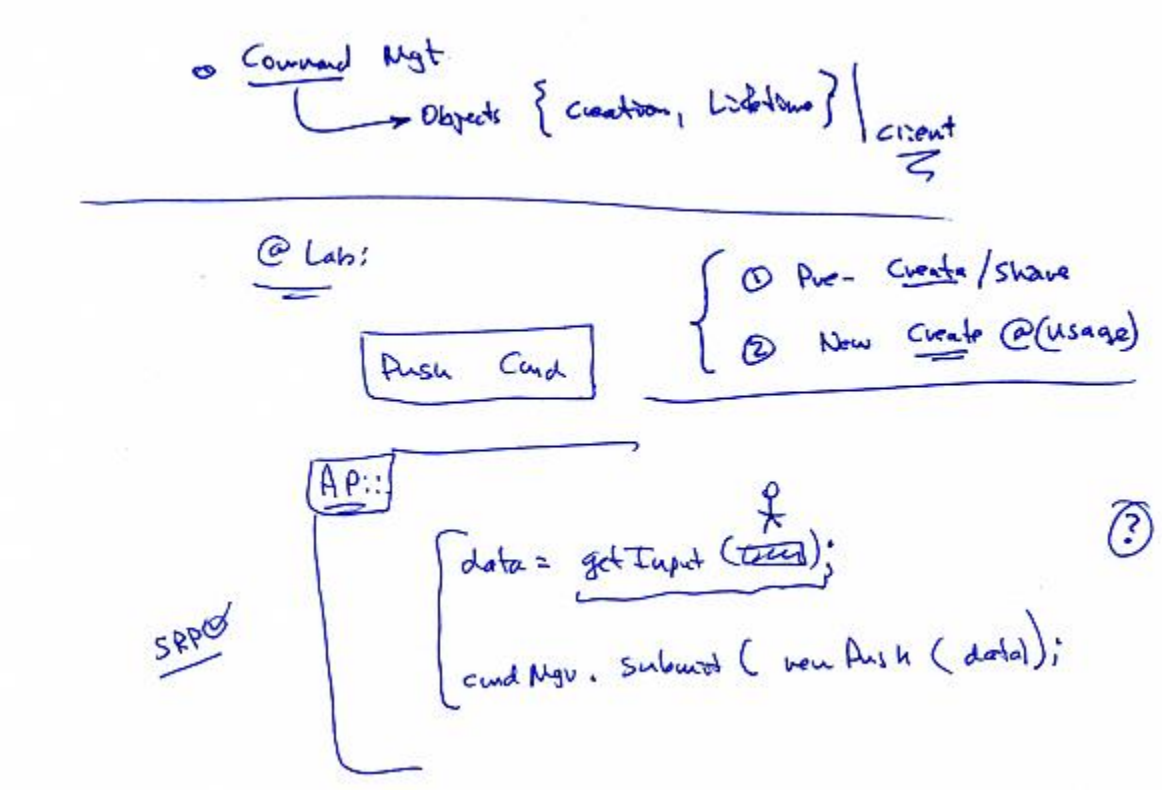
- Example code…
  - GUI
  - Switch …
  - Scheduler
  - ⇨ different *CmdManager* models /methods
    - Event driven (swing)
    - Message driven (switch)
    - Timer driven (scheduler)
    - …
- *CommandButtons*
  - Combine functions: GUI & Command
    (Only works with stateless commands)



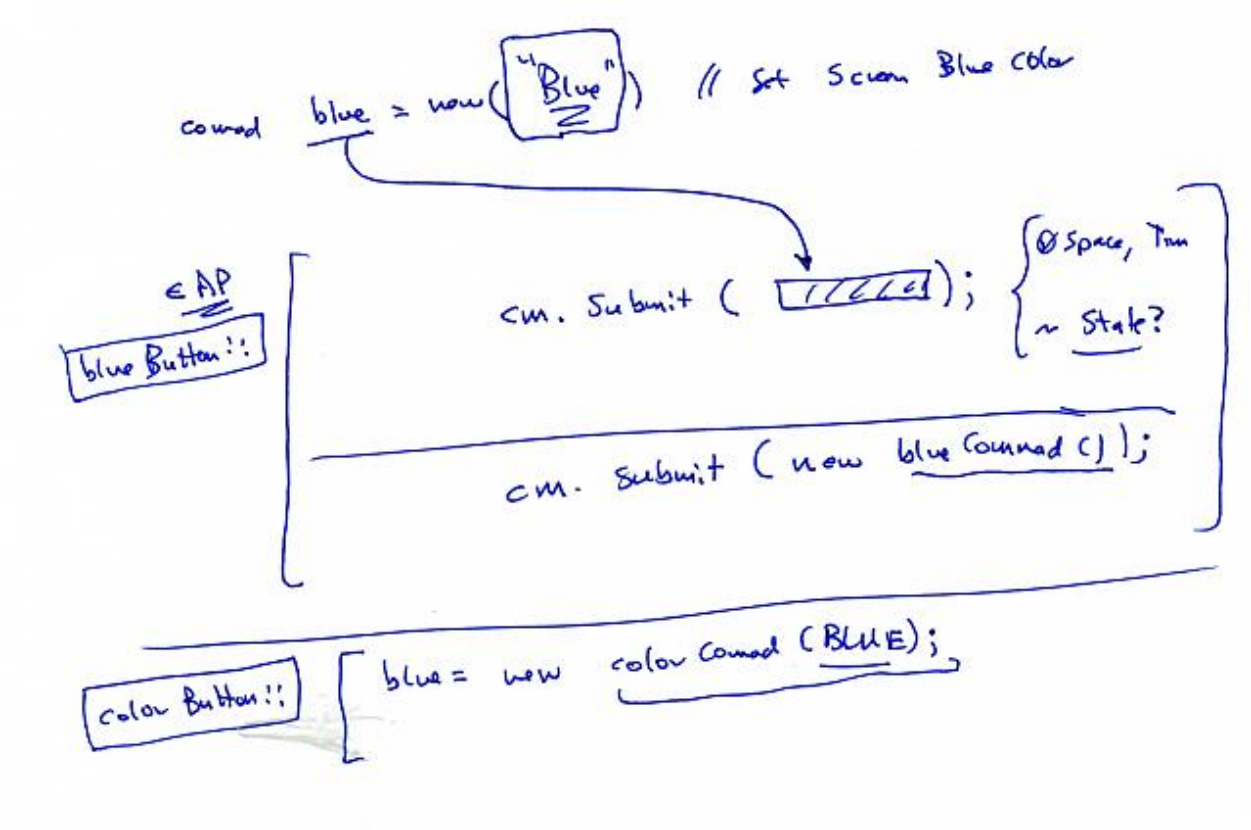
- Swing undo/redo
- Lab: Φ_2
  - Review ϕ_1
  - Add Command Manager
  - Stateful commands { *undo, redo* }
  - *meta*-commands ?
  - GUI *mediator*



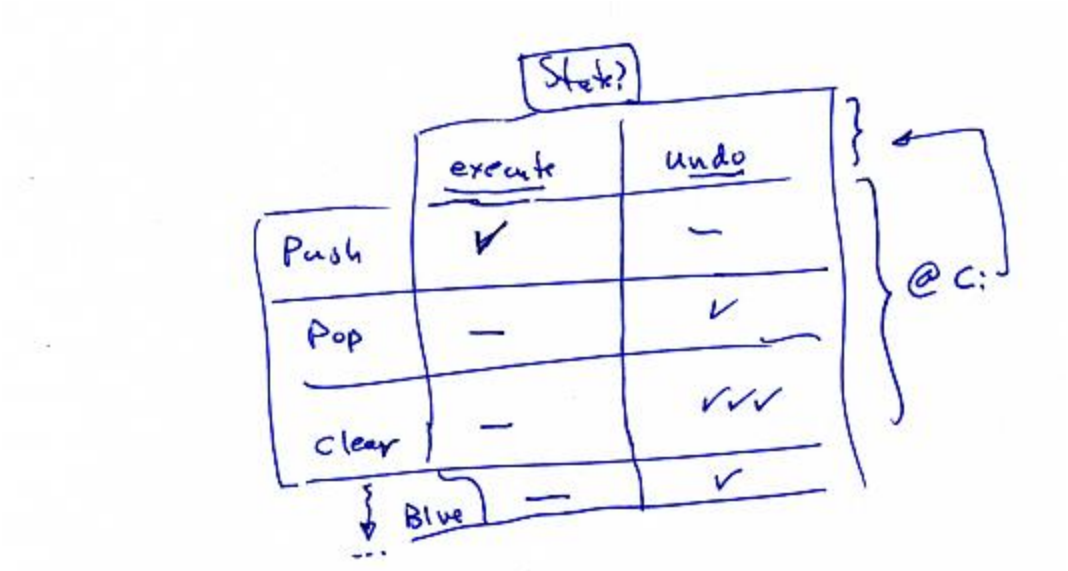
Review of Command Lab – Φ_2



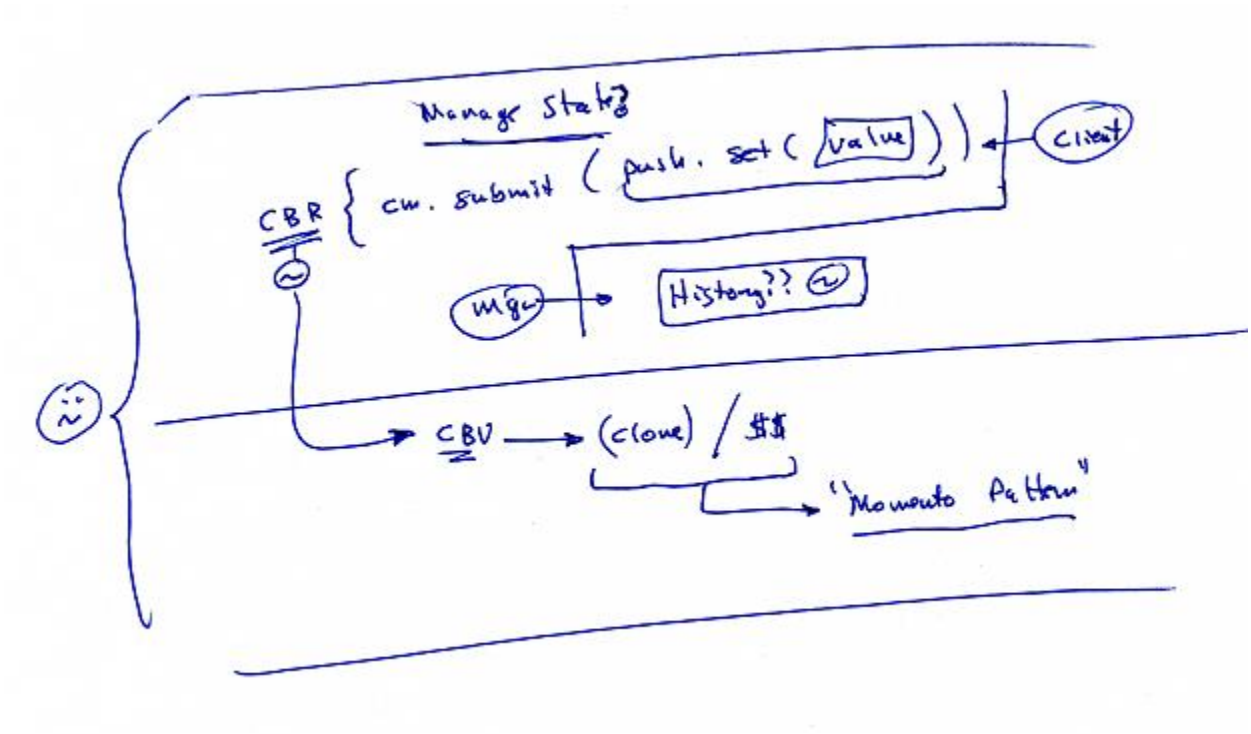
- Design issue of Command object lifetimes and management



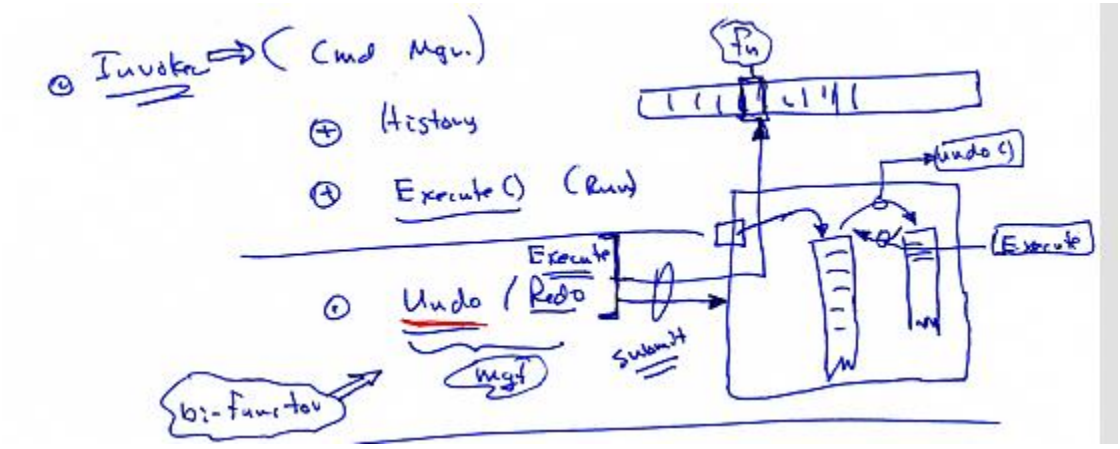
- Example – Simple (stateless) command



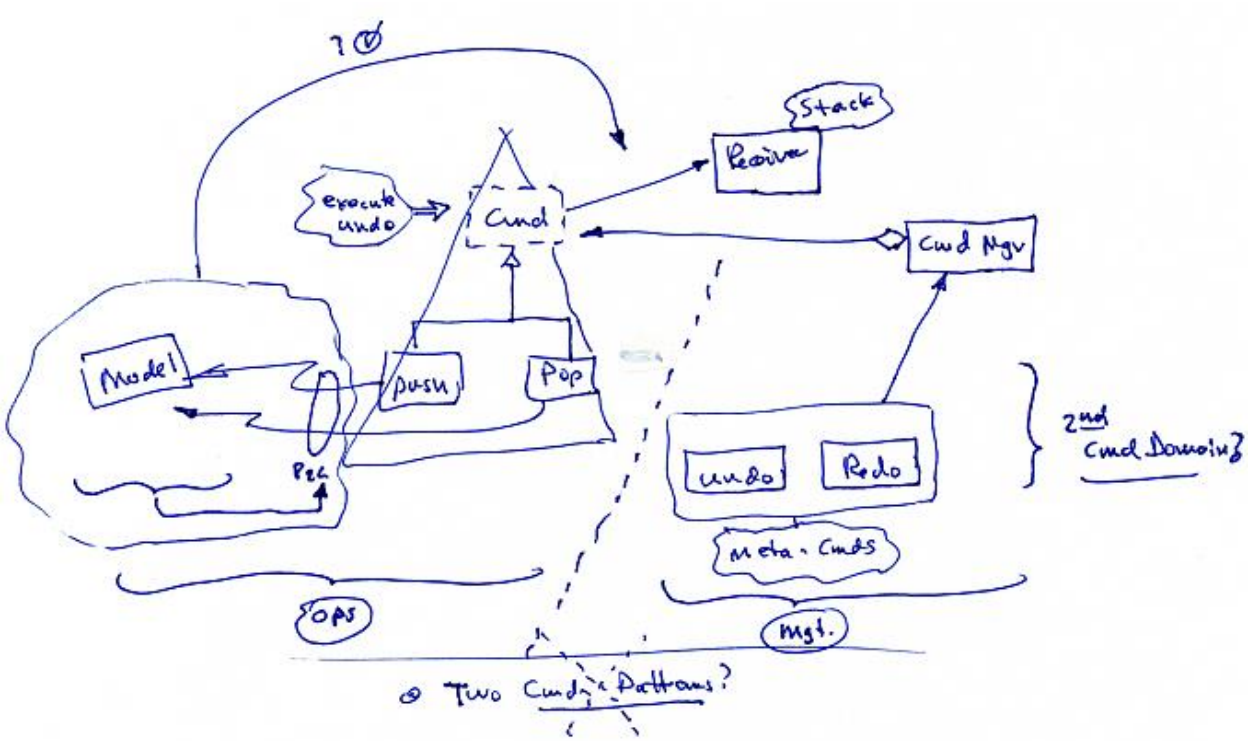
- Stateful commands



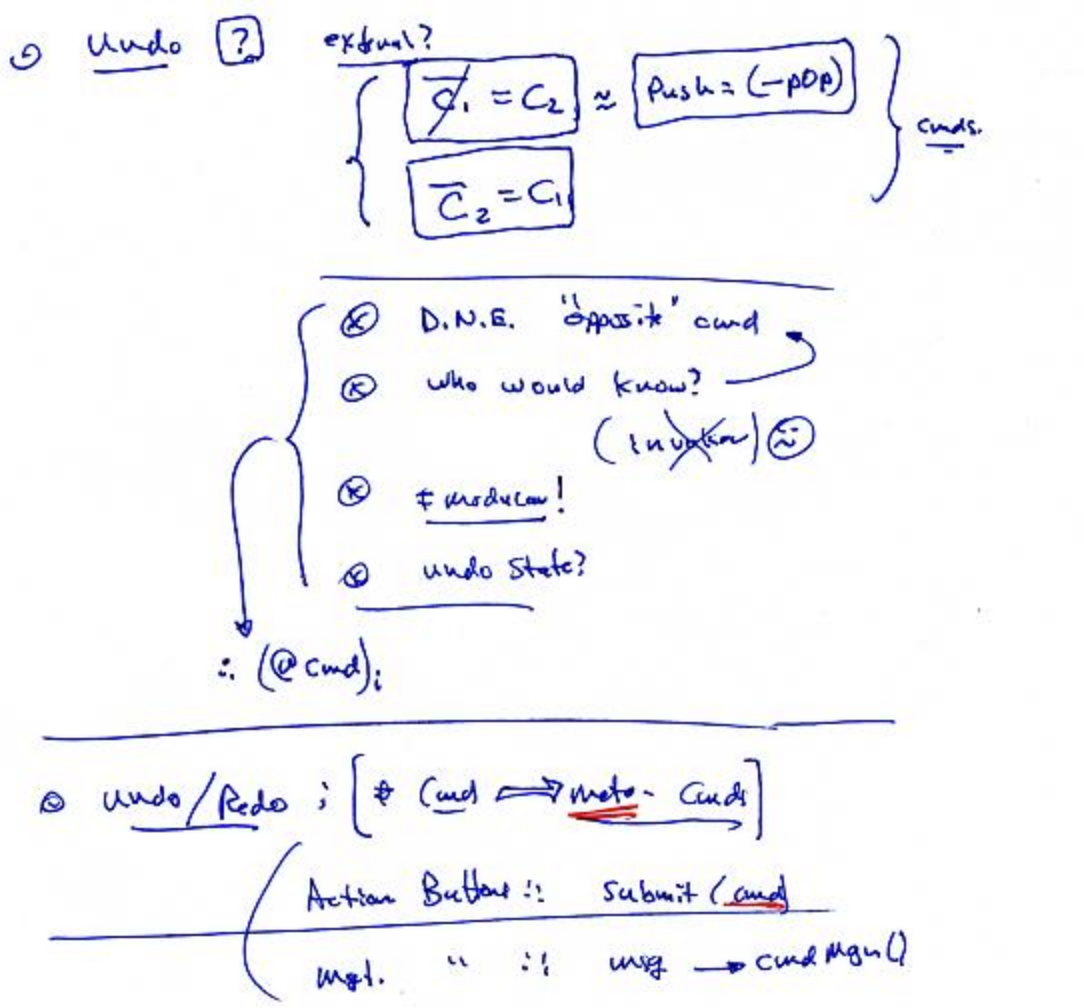
- How to manage state in commands?



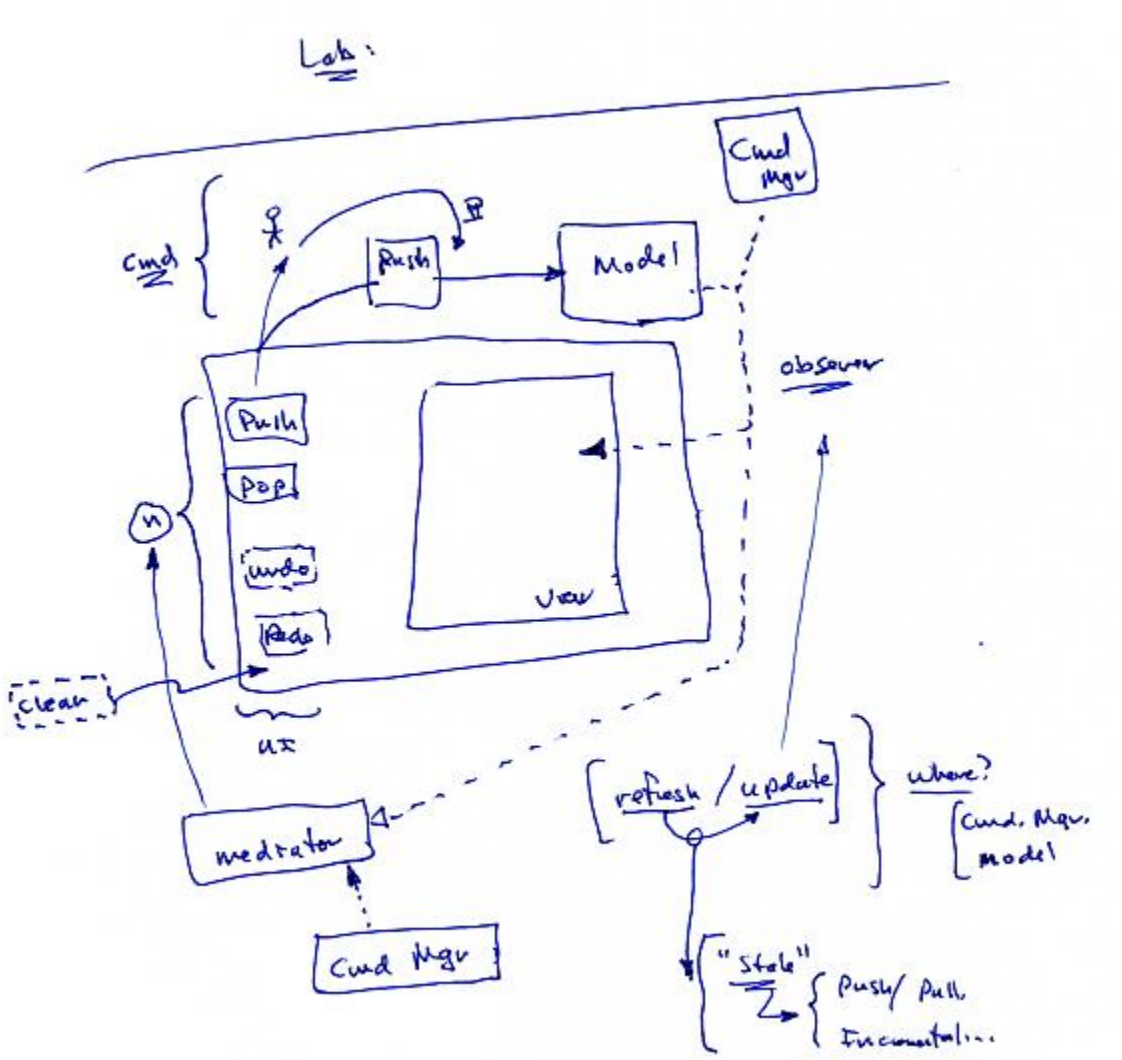
- Converting invoker to Command-manager



- Overall architecture of Lab



- 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. ☹

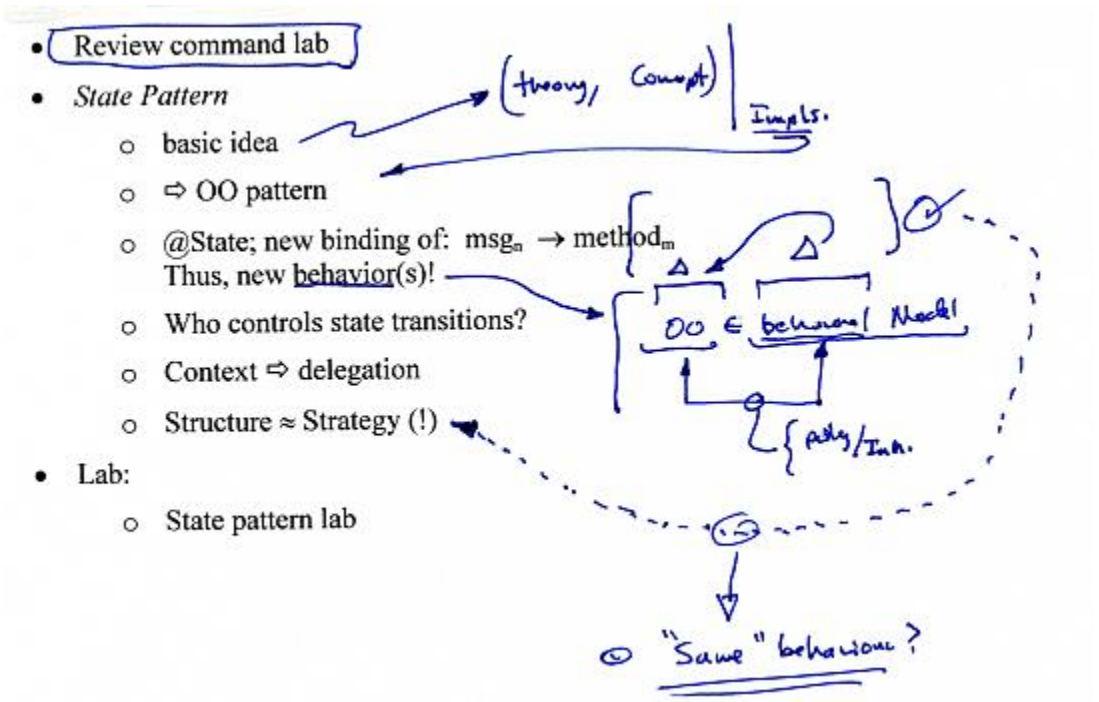
- Swing supports undo/redo for its standard components

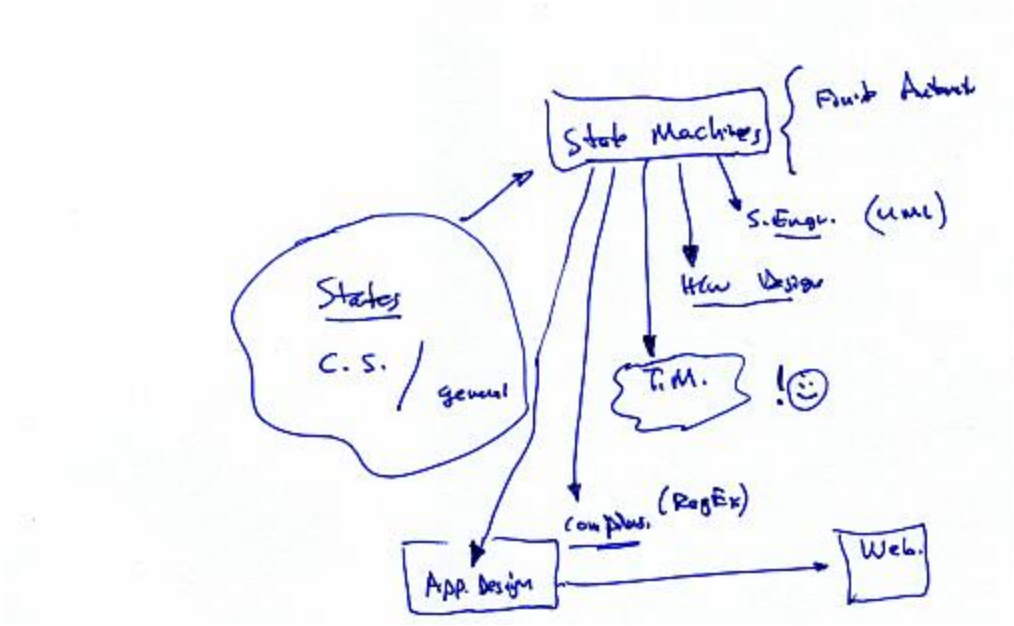
```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.undo.*;
import javax.swing.text.*;
import javax.swing.event.*;

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();
        }
```

- Swing Example

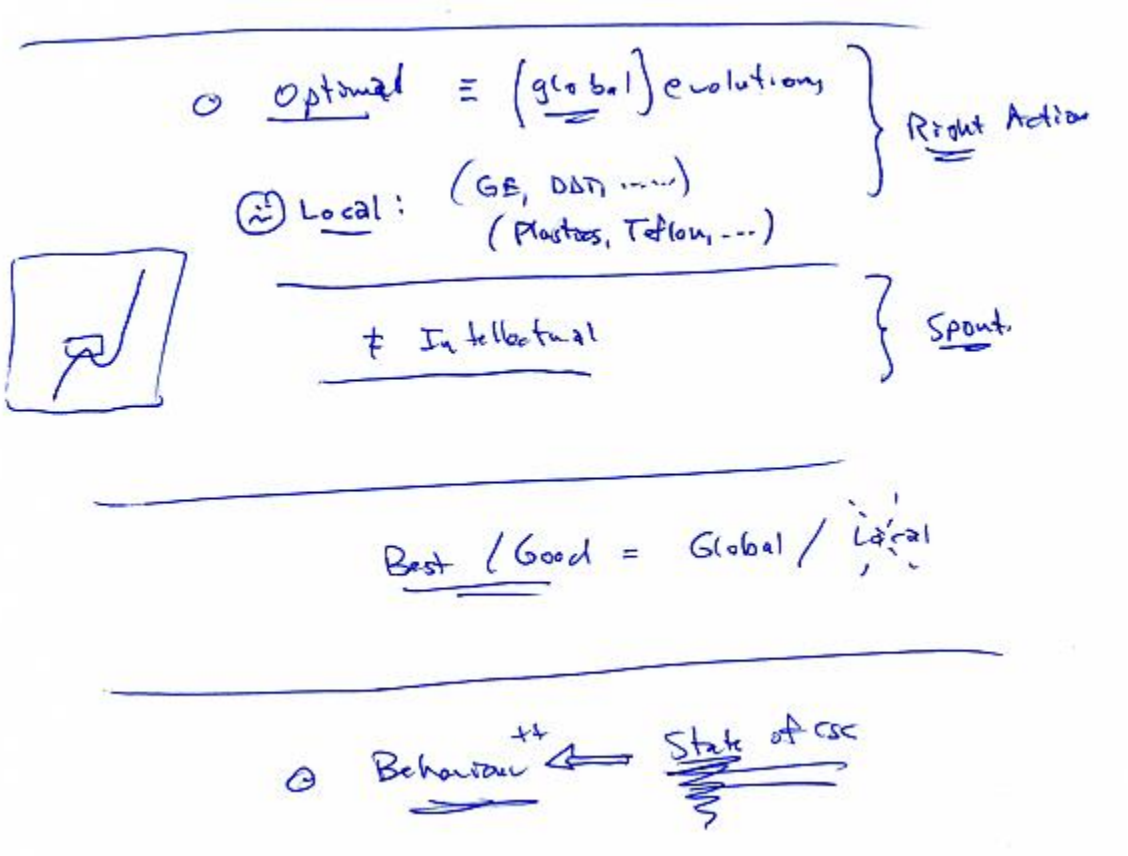
Day 10: State Pattern

- Review command lab
- *State Pattern*
  - basic idea
  - ⇨ OO pattern
  - @State; new binding of: $msg_n \rightarrow method_m$
    Thus, new behavior(s)!
  - Who controls state transitions?
  - Context ⇨ delegation
  - Structure ≈ Strategy (!)
- Lab:
  - State pattern lab

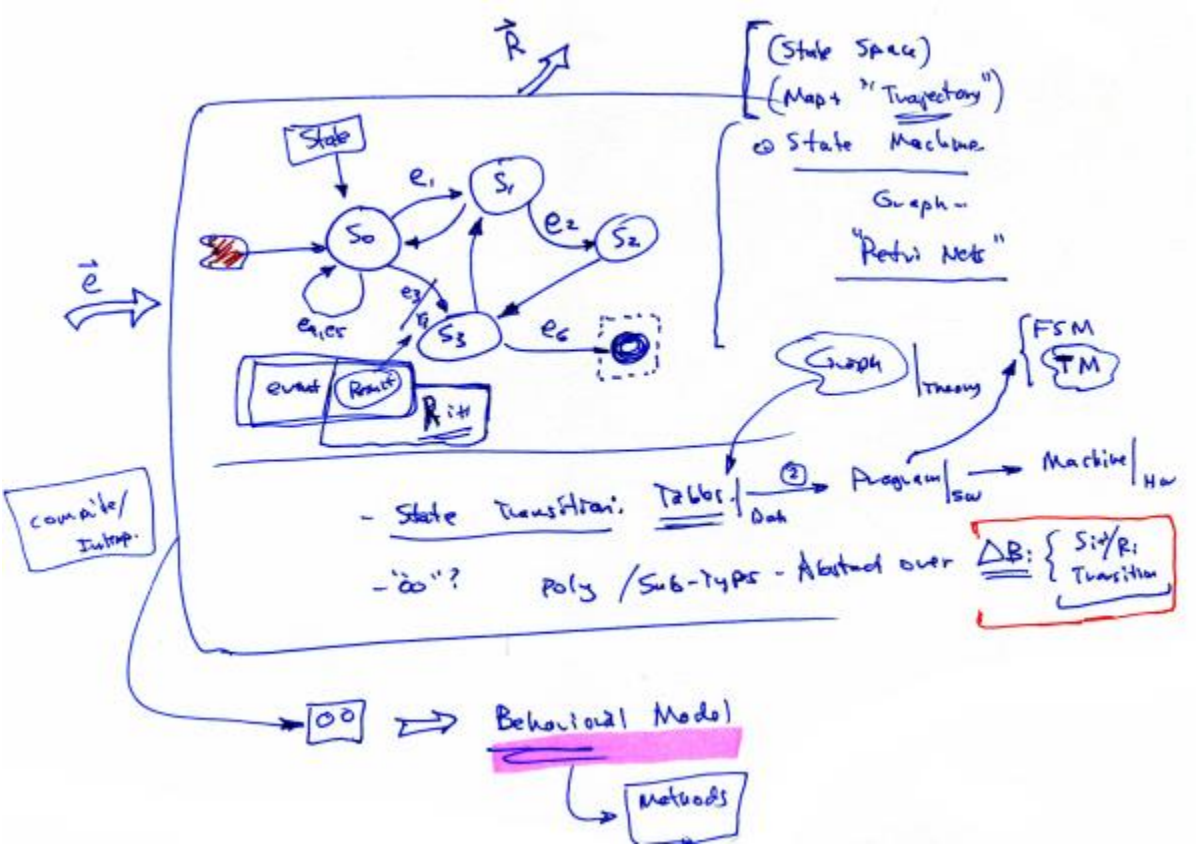




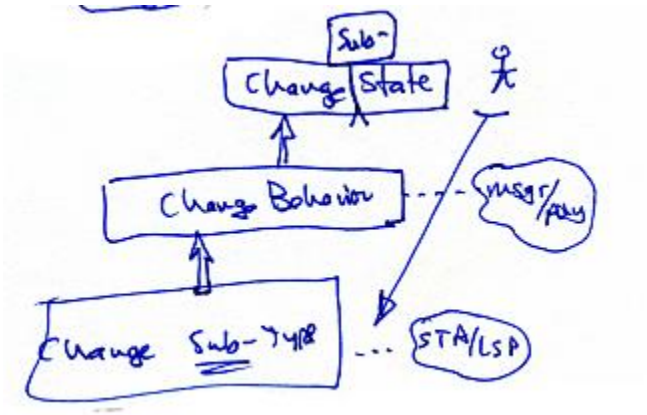
- State machines are a general idea in Computer Science



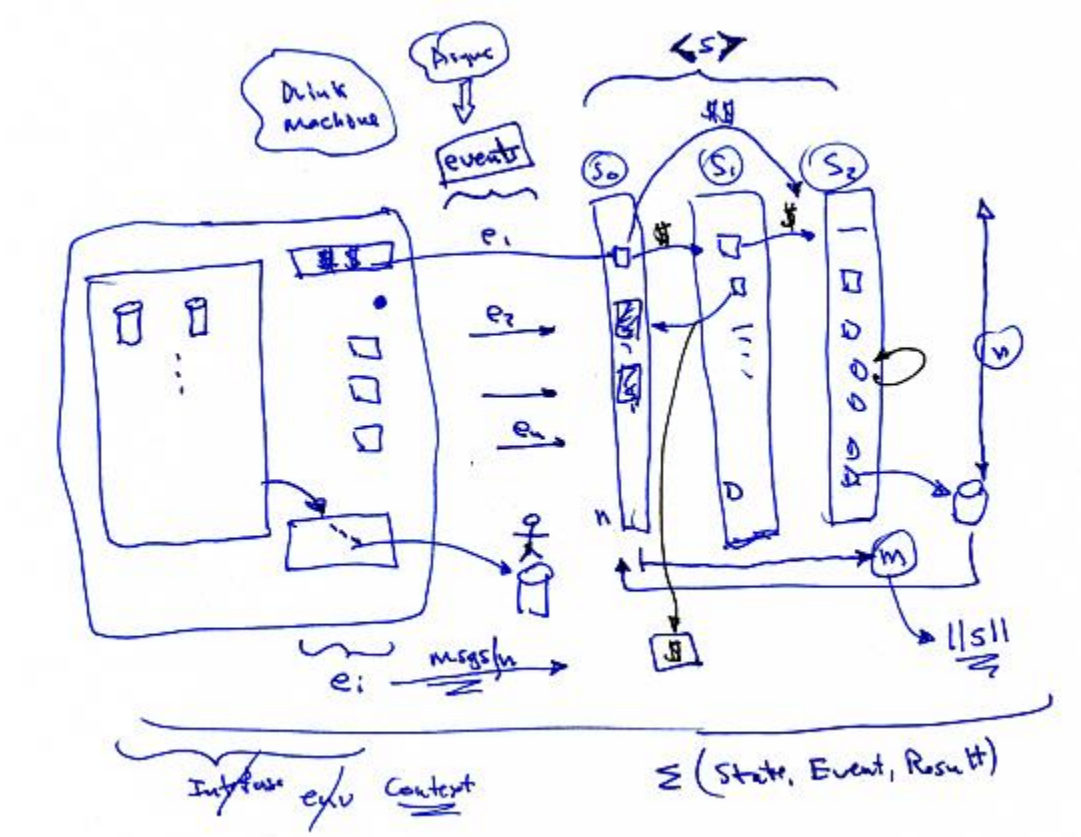
- Optimal behavior (*Right Action*) is not intellectual, only possible from higher states of consciousness



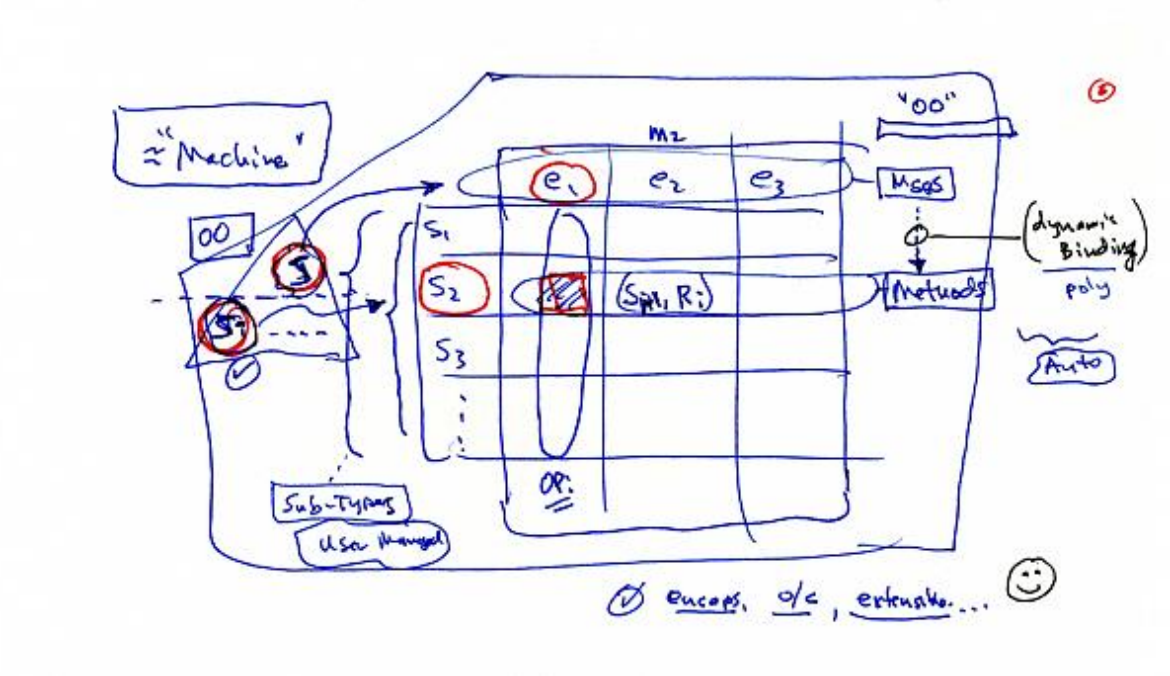
- State space graphs are a general model



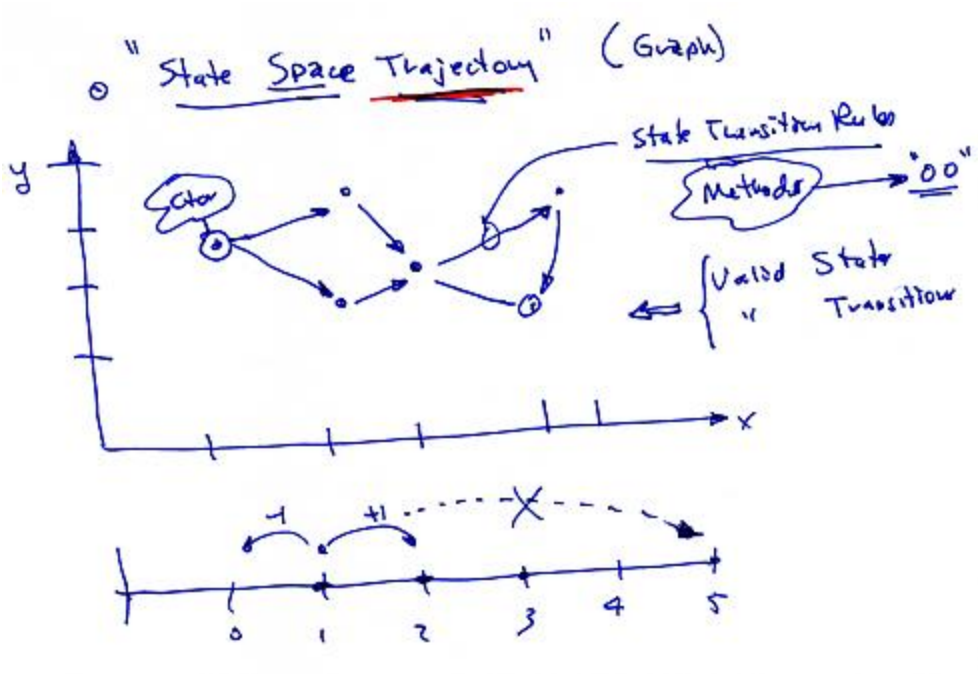
- The OO model implementation



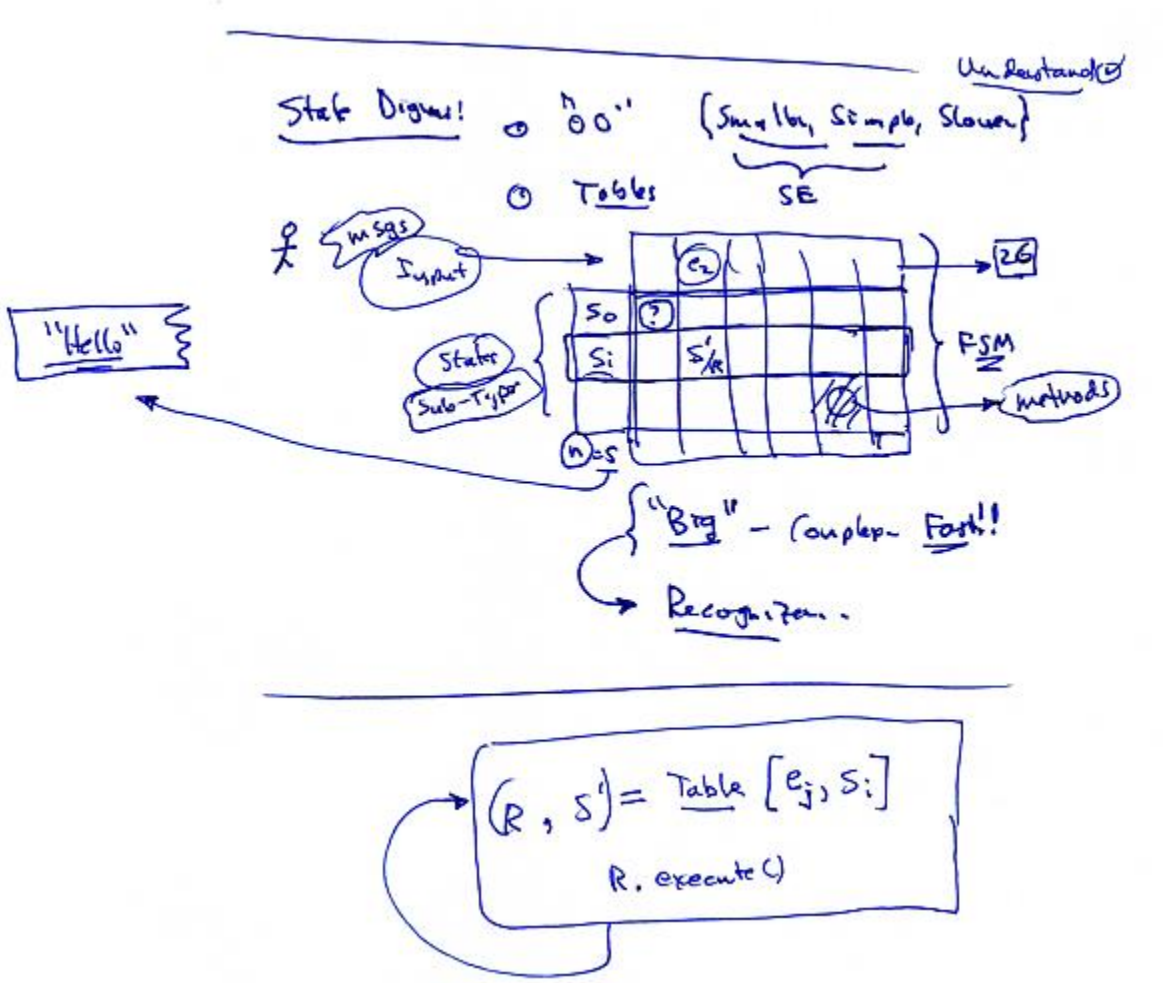
- Example: A Vedic (organic) juice machine (!)



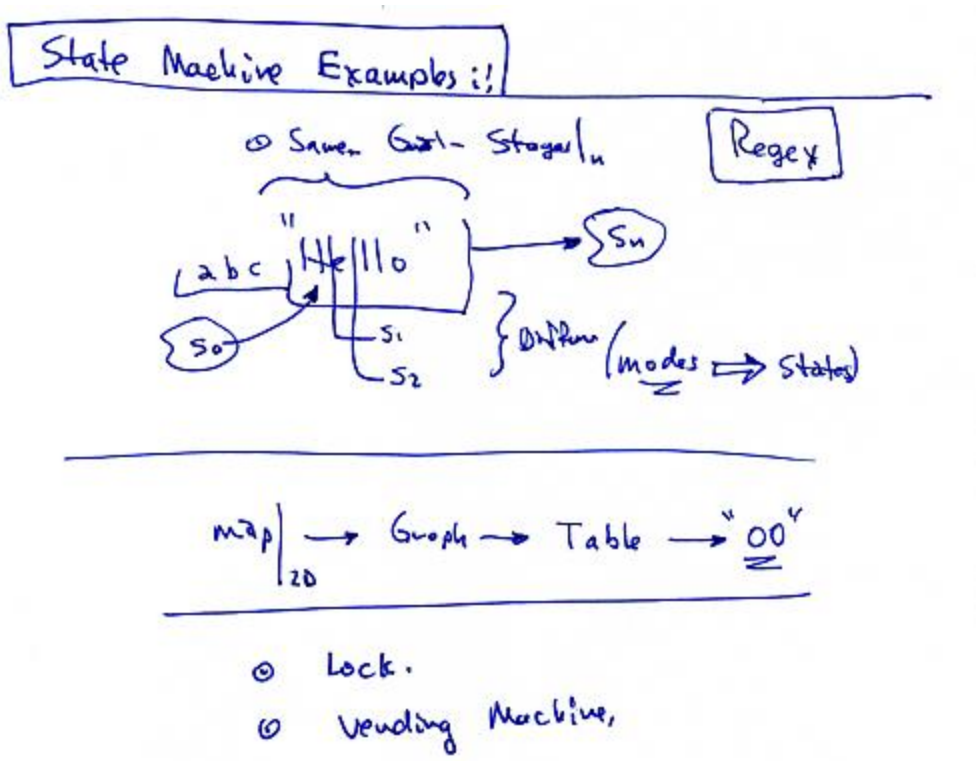
- One can specify the state model in a table, and then map that to an OO model



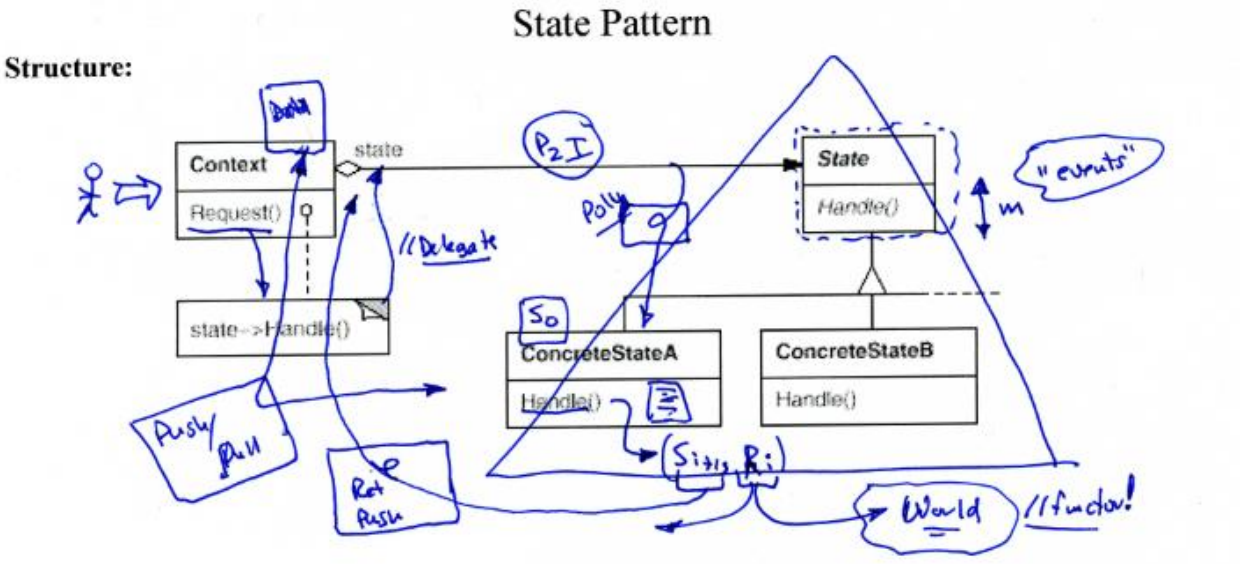
- The trajectory describes the valid state transitions
- Other points or paths may be possible, but not valid (legal)



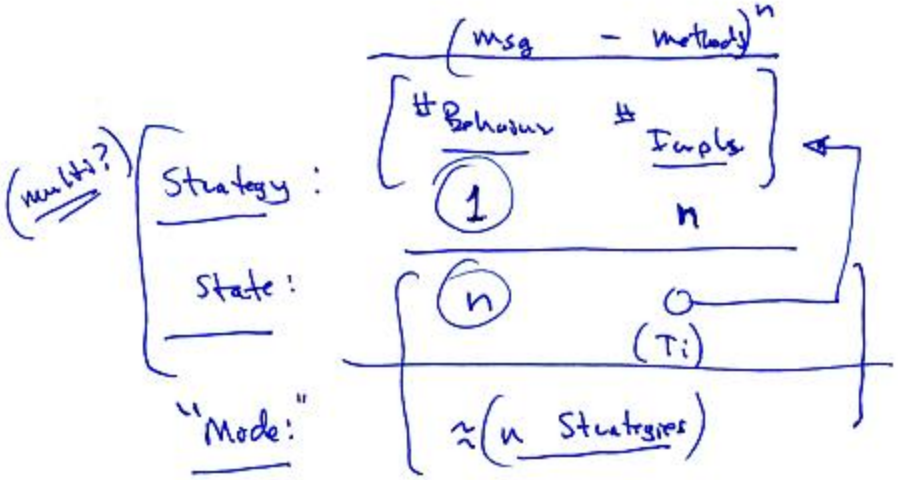
- For more complex (big) applications, a table driven state machine would be faster, but harder to understand and create
- Thus there are tools to do this automatically for those applications (compilers, lexers, Regex, …)



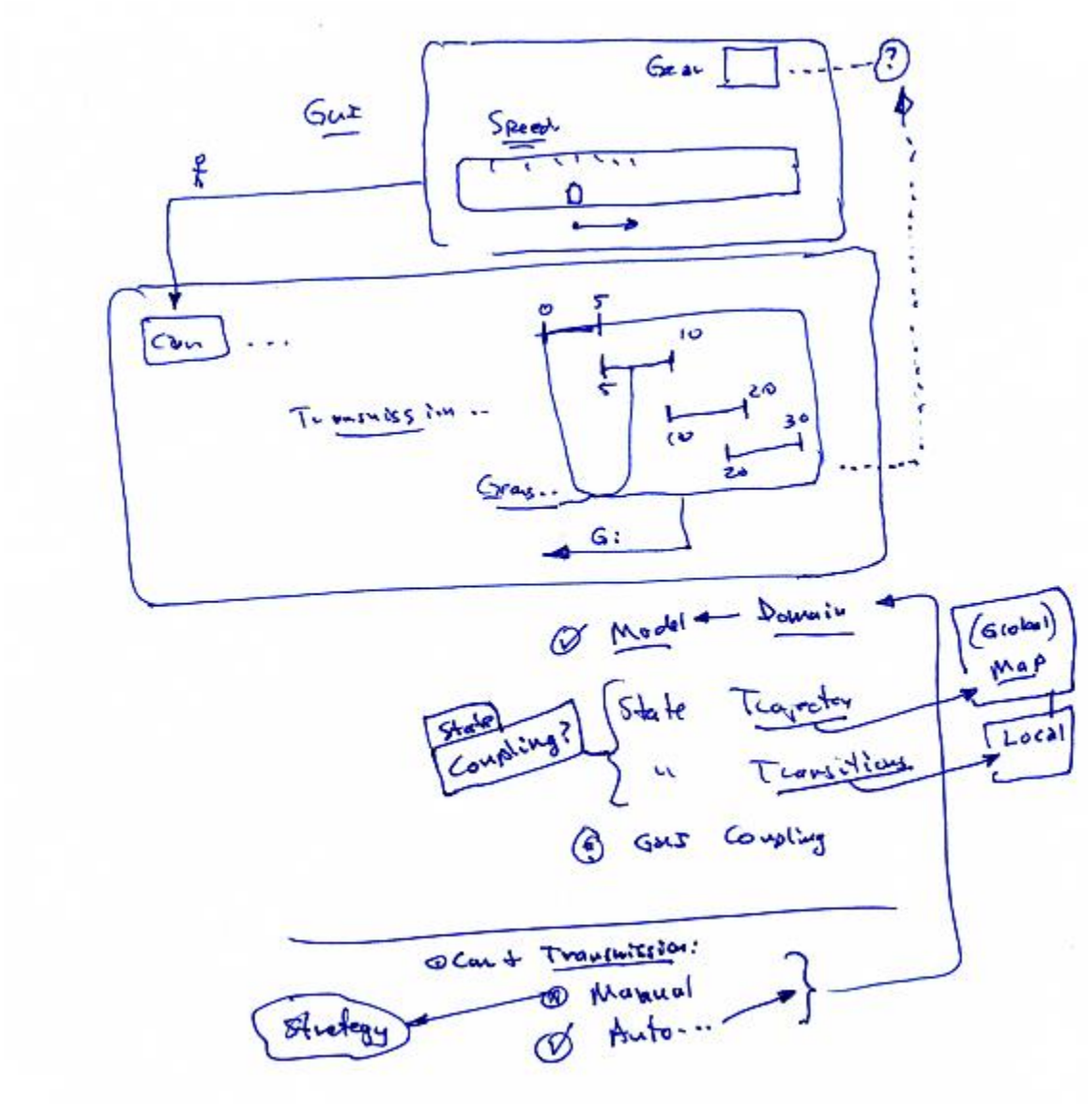
- Examples of state machines



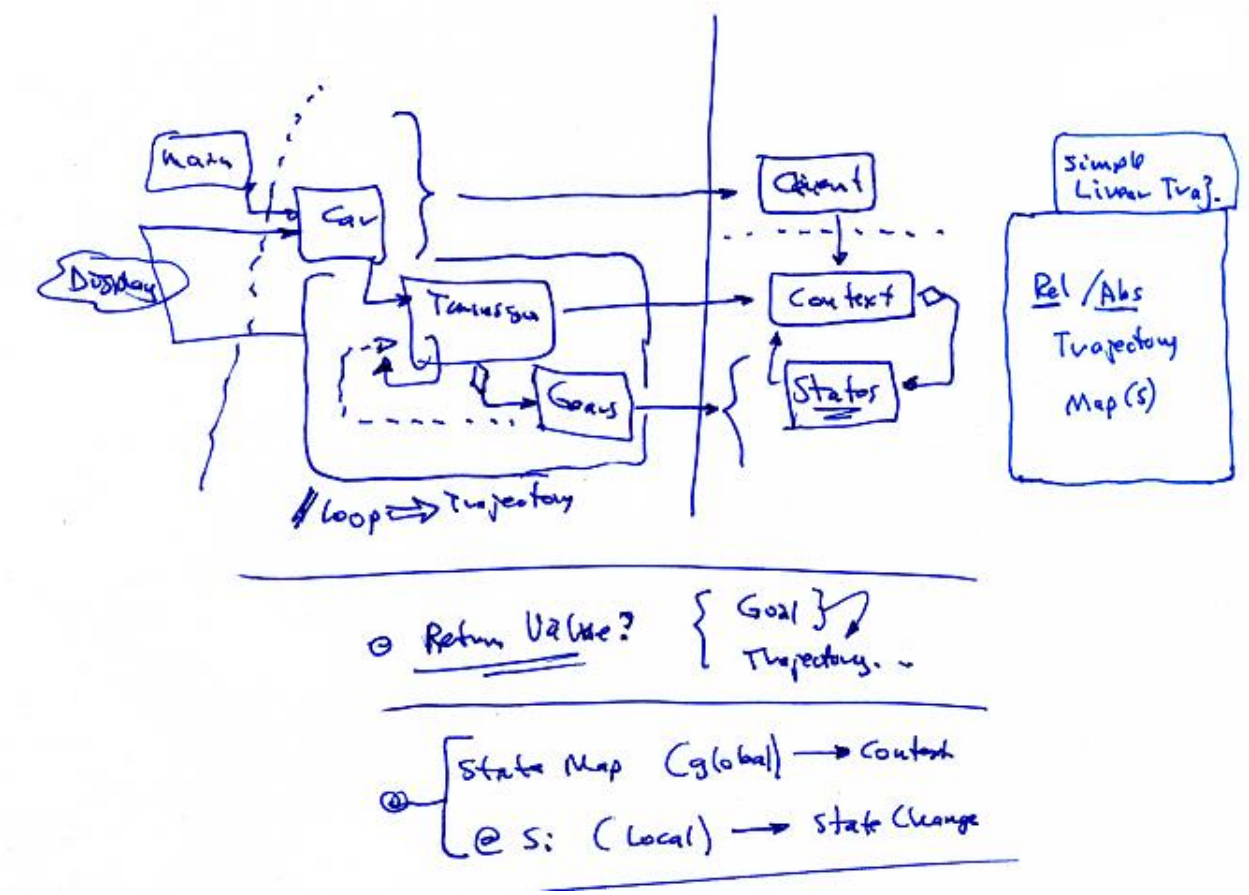
- GOF model example



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

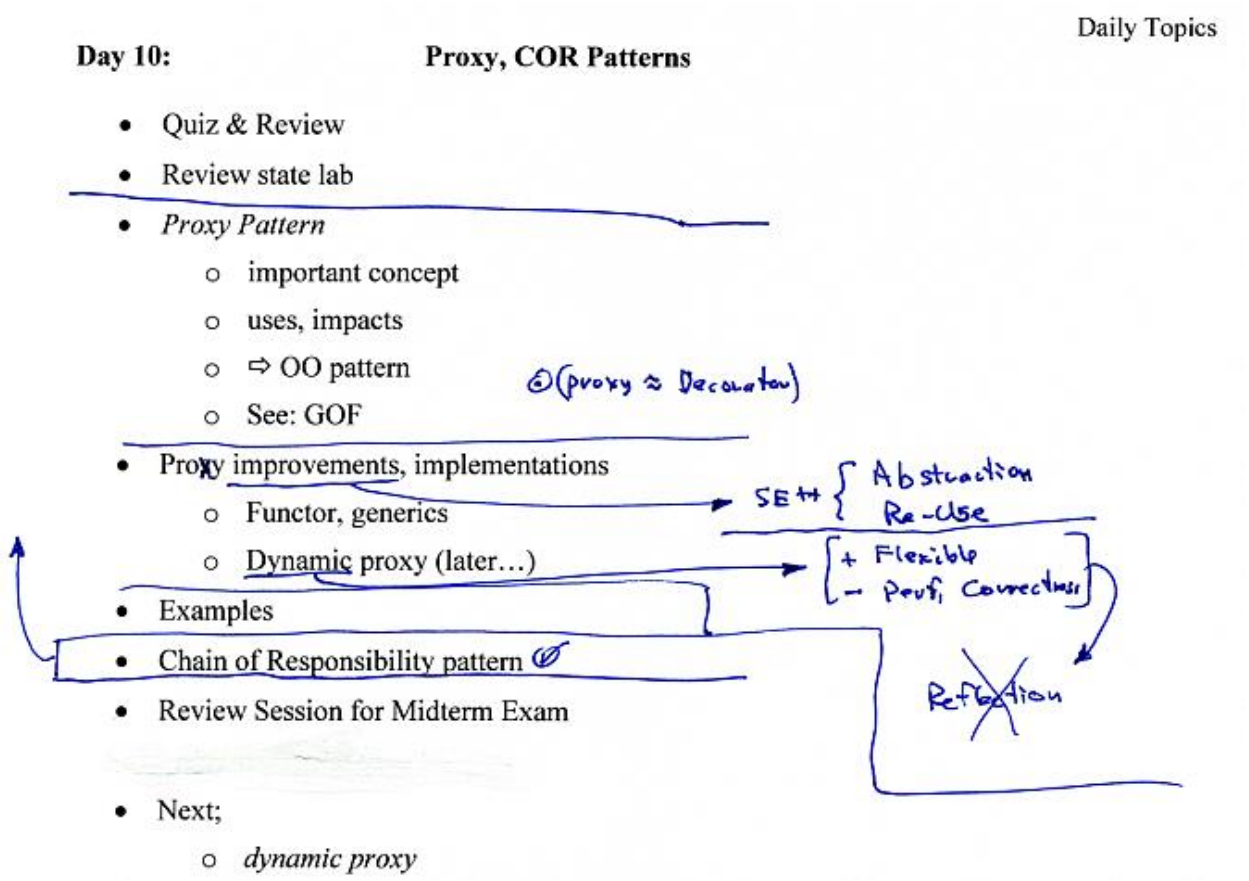


- State pattern lab



- State pattern lab review

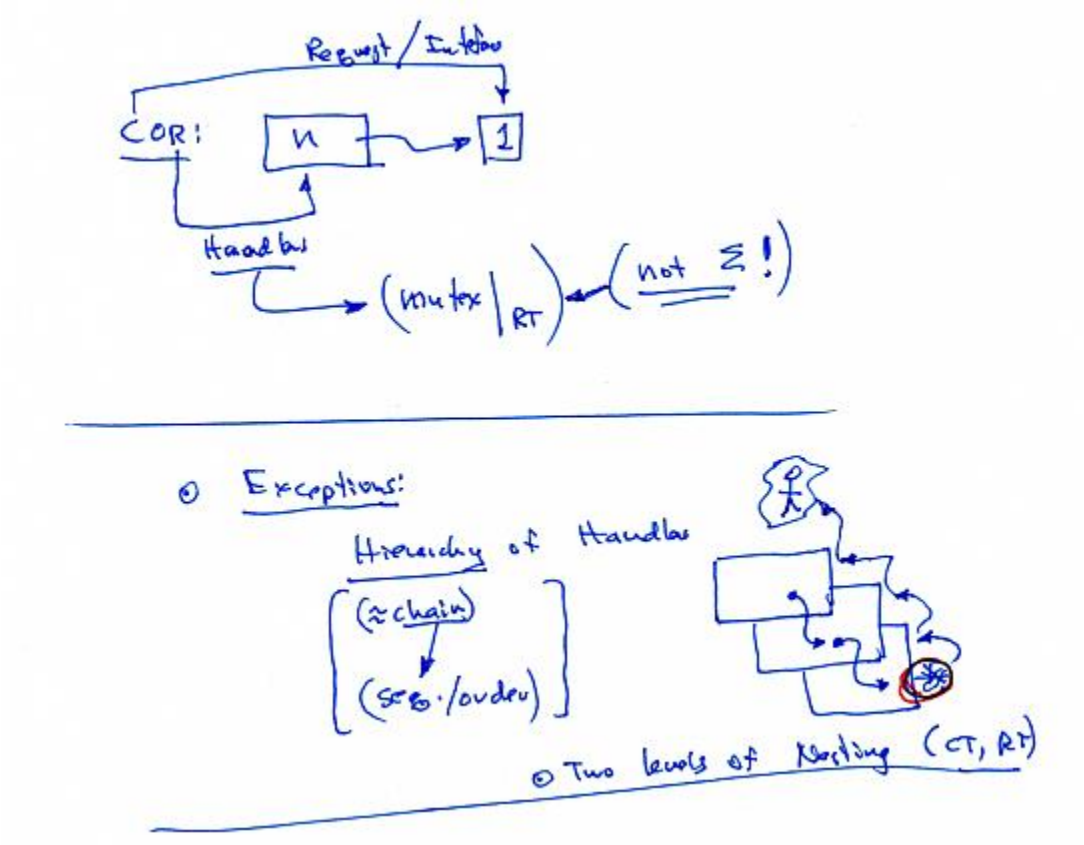
Day 11: Proxy & COR Pattern



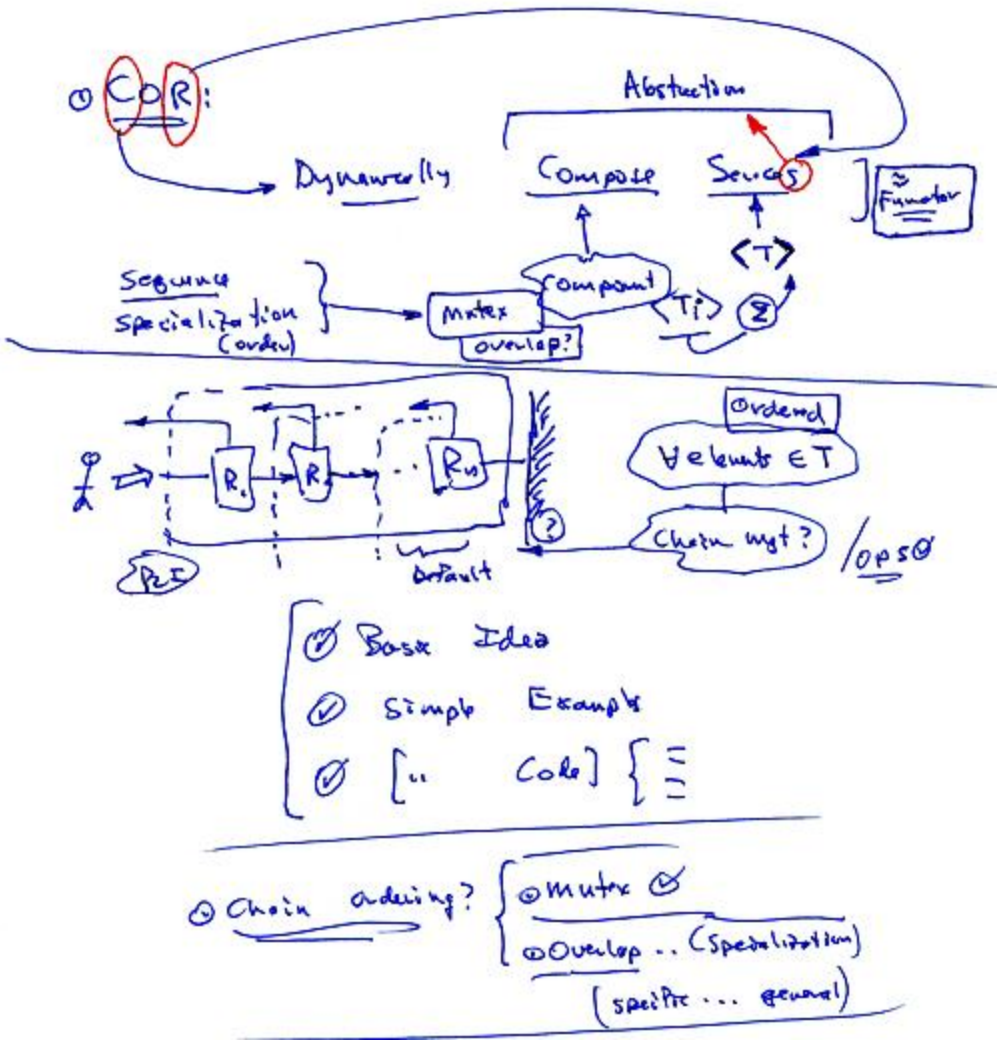
Daily Topics

Day 10: **Proxy, COR Patterns**

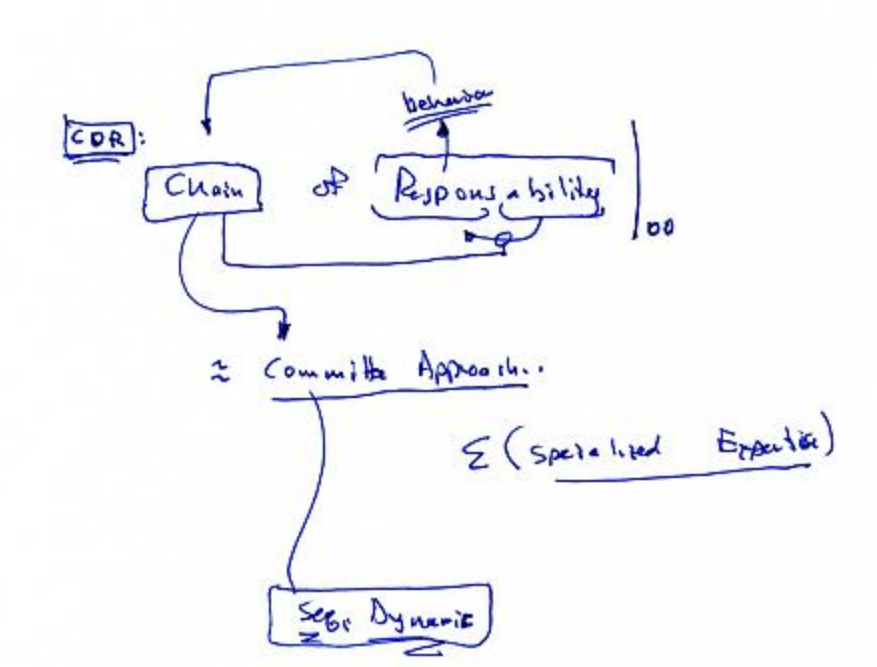
- Quiz & Review
- Review state lab
- *Proxy Pattern*
  - important concept
  - uses, impacts
  - ⇨ OO pattern
  - See: GOF
- Proxy improvements, implementations
  - Functor, generics
  - Dynamic proxy (later…)
- Examples
- Chain of Responsibility pattern
- Review Session for Midterm Exam
- Next;
  -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



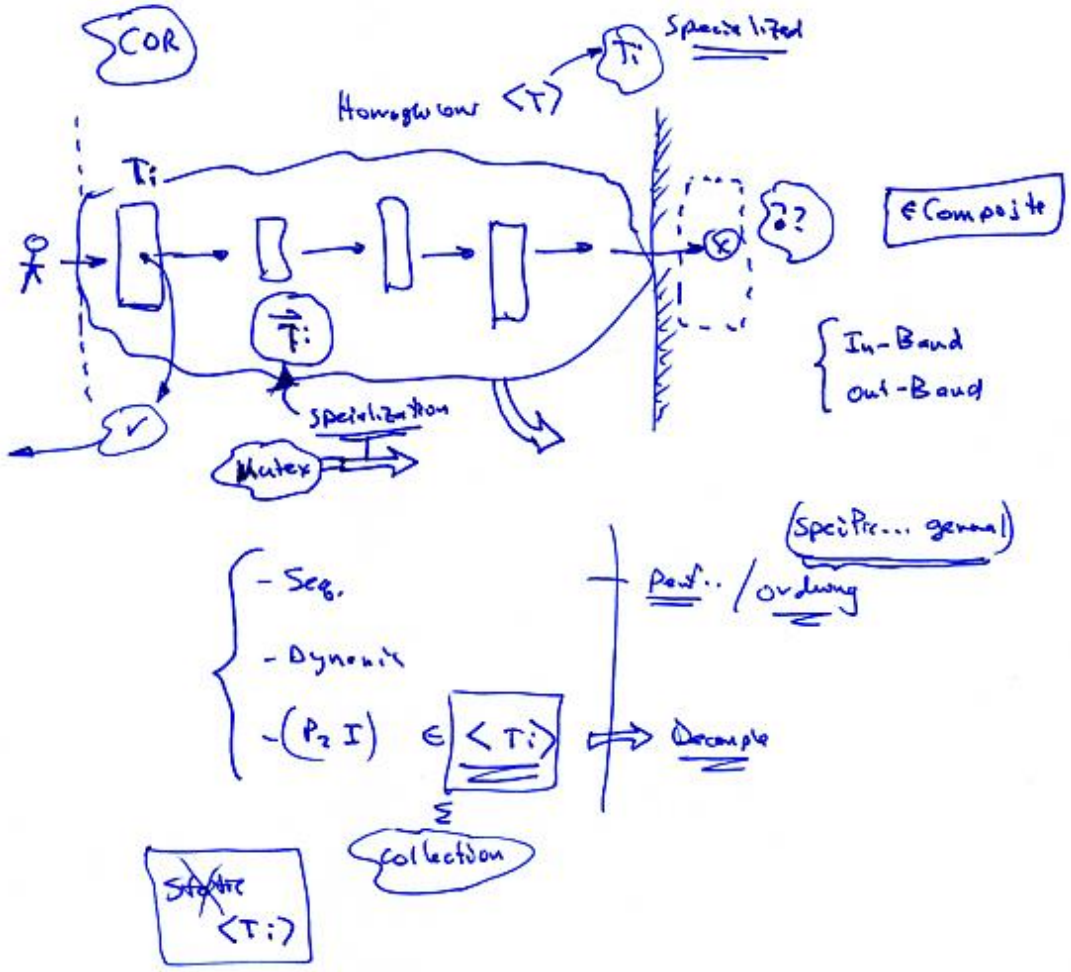
- COR pattern is already familiar, Exception hierarchies
- They are linear, hierarchical (ordered)
- Also they have both static (CT, definitions scopes) and dynamic (RT, function call hierarchies) structure



- Basic idea of COR pattern


COR:
Chain
of
Responsability
Behaviour
Committe Approach..
Σ (specialized Expertise)
Segr. Dynamic

- COR ideas



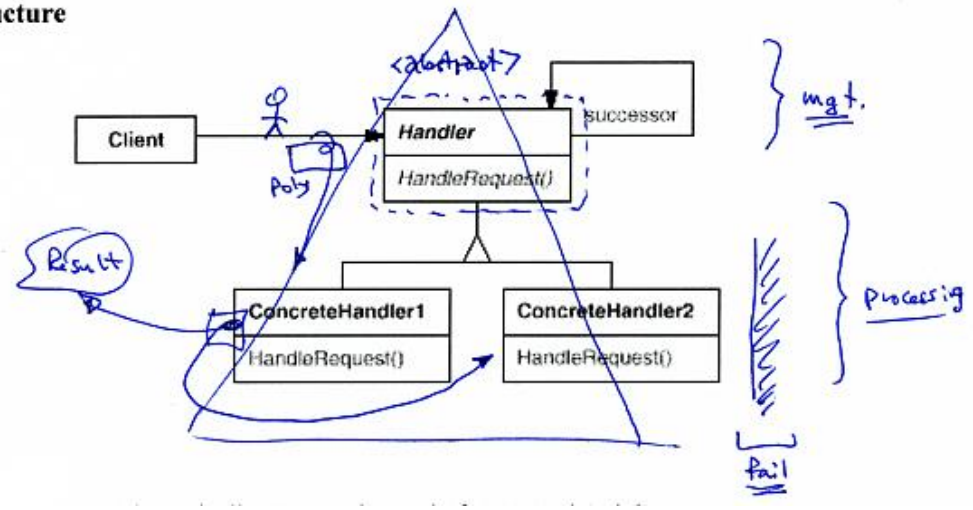
- COR Structure

Chain of Responsibility

Intent

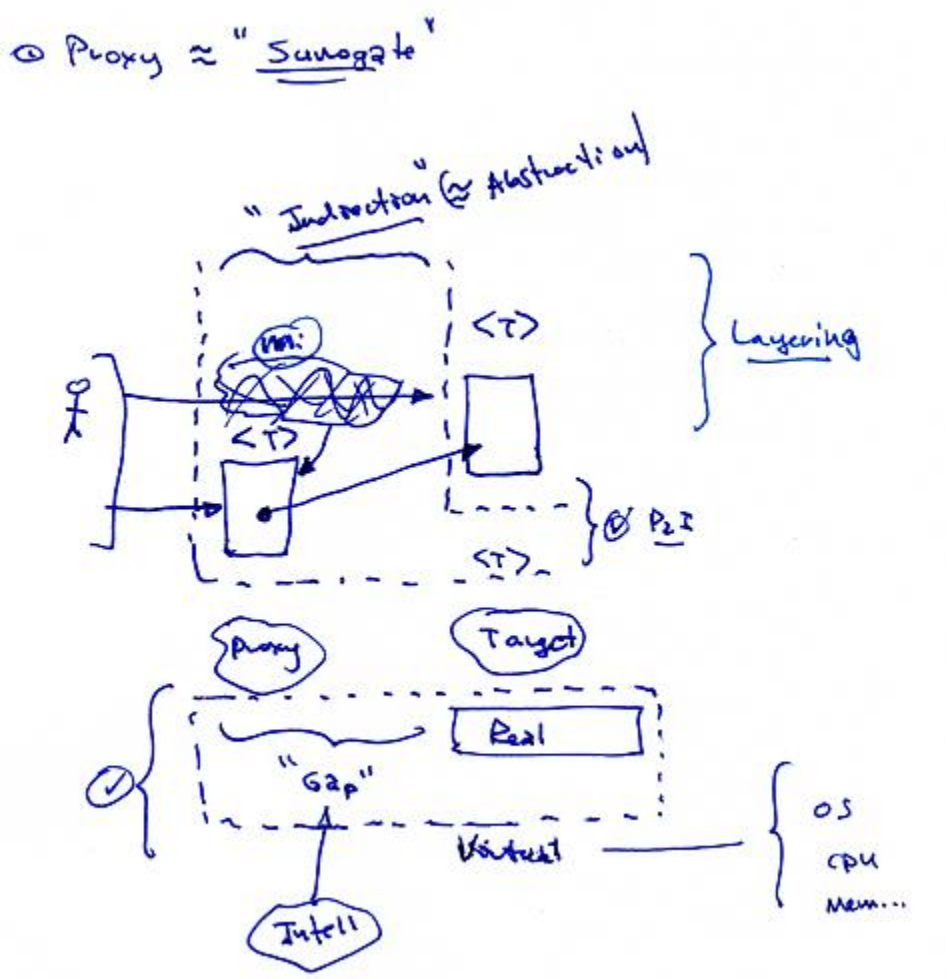
Decouple the sender of a request and its receiver by creating a pool of potential receivers, and giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Structure

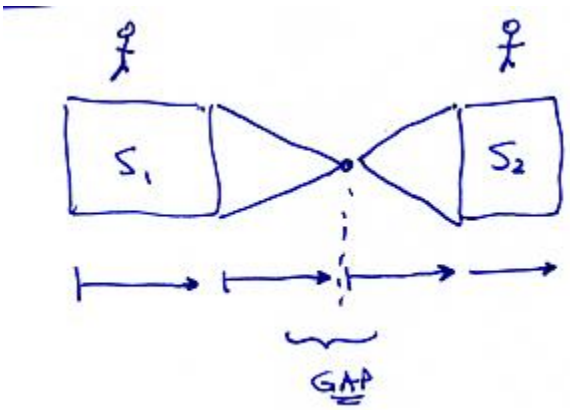


- GOF description

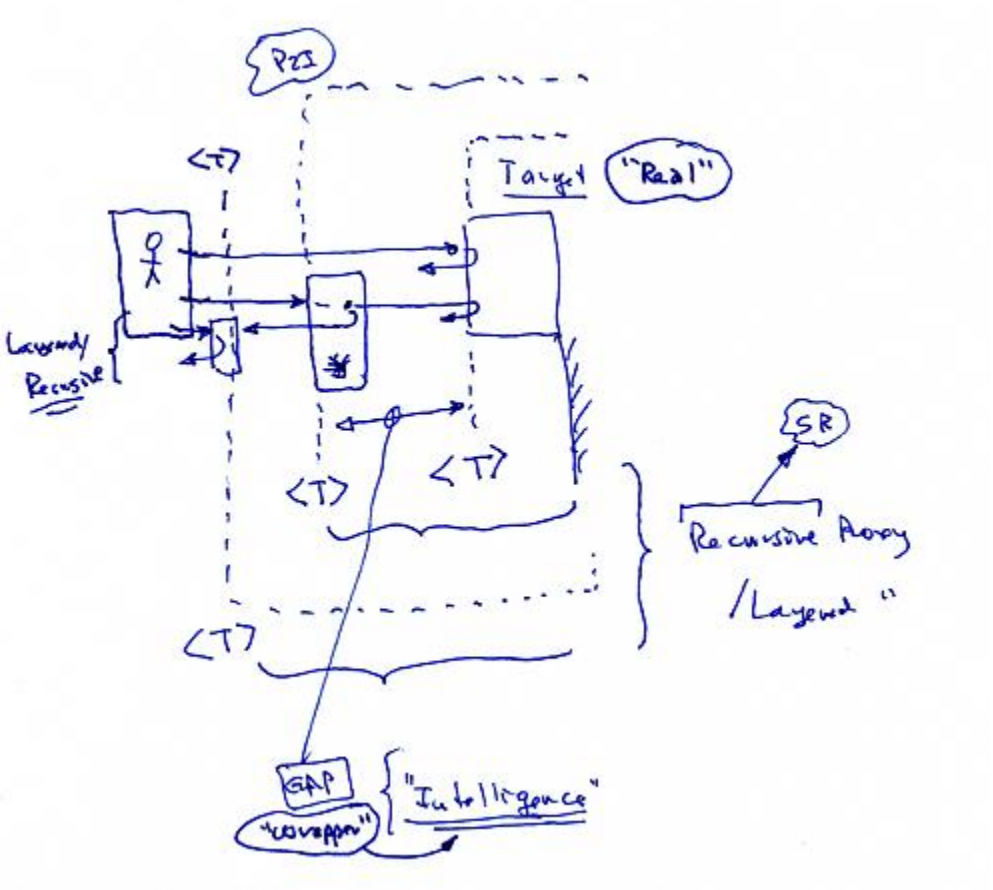
Proxy Pattern::



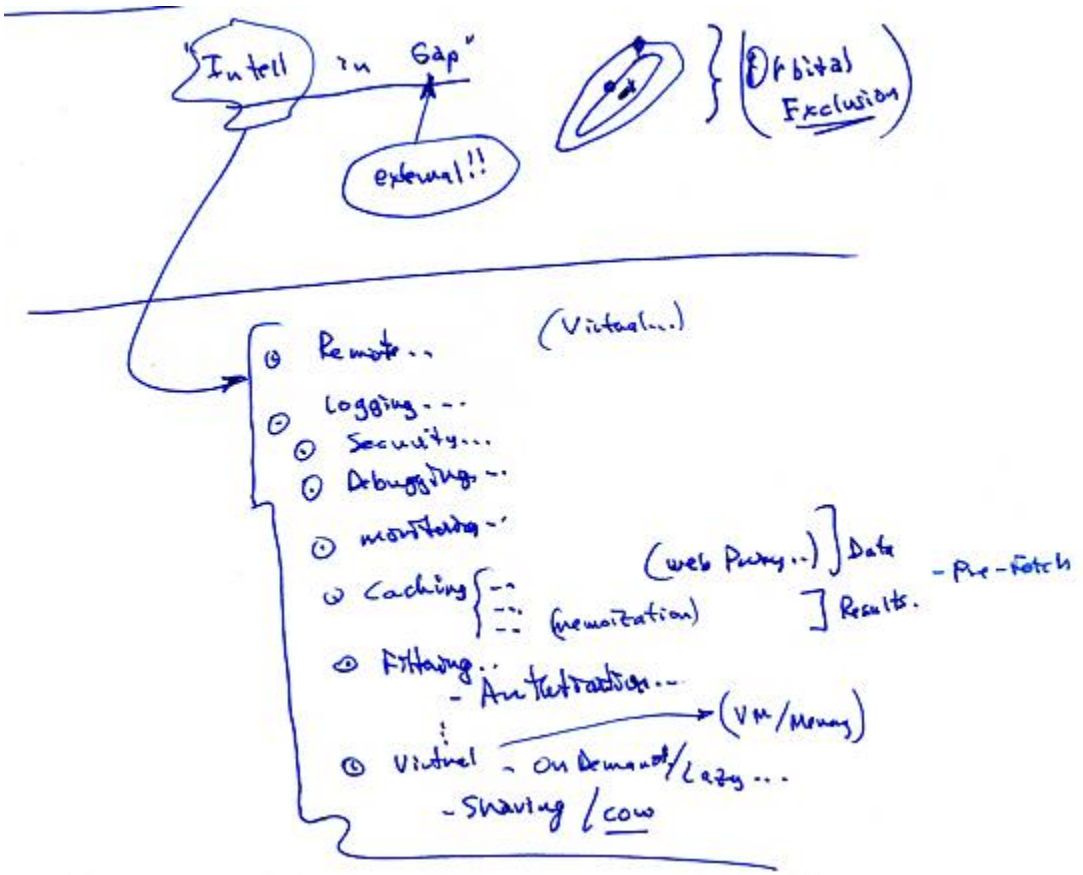
- General idea of Proxy Pattern



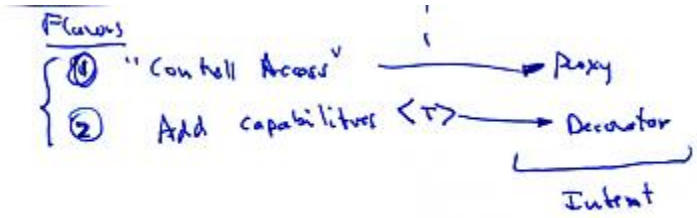
- SCI describes the model of the transcendental "*intelligence in the gap*"



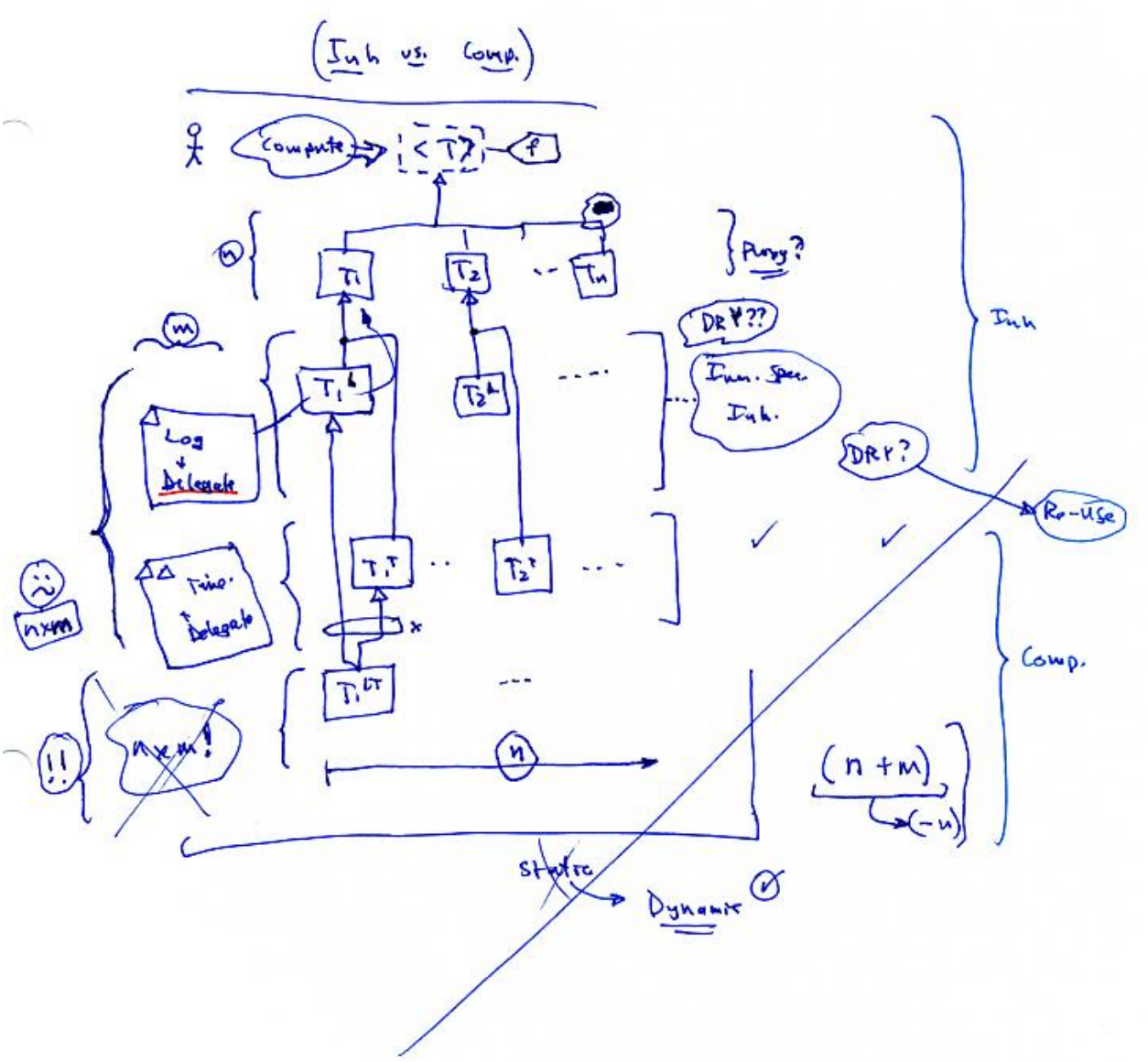
- Model of the proxy pattern



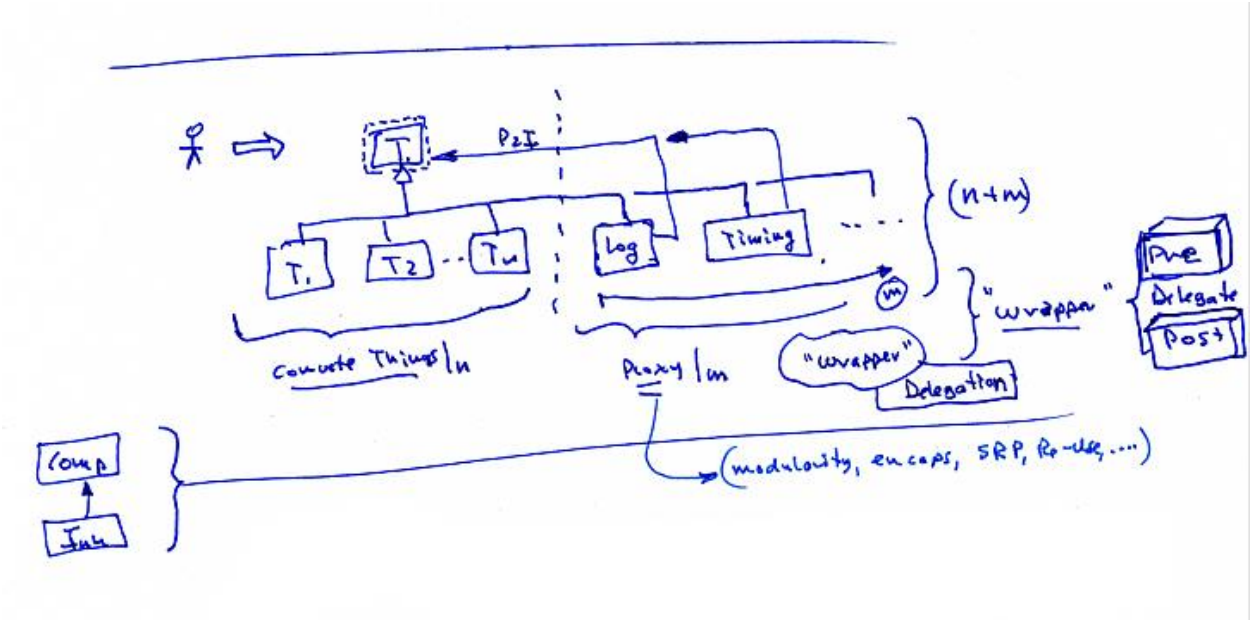
- Applications of Proxy



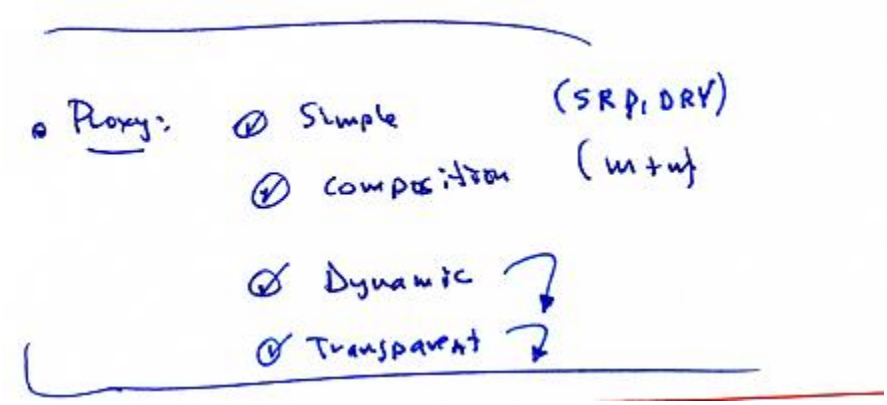
- Decorator is related to proxy; same structure, different intent



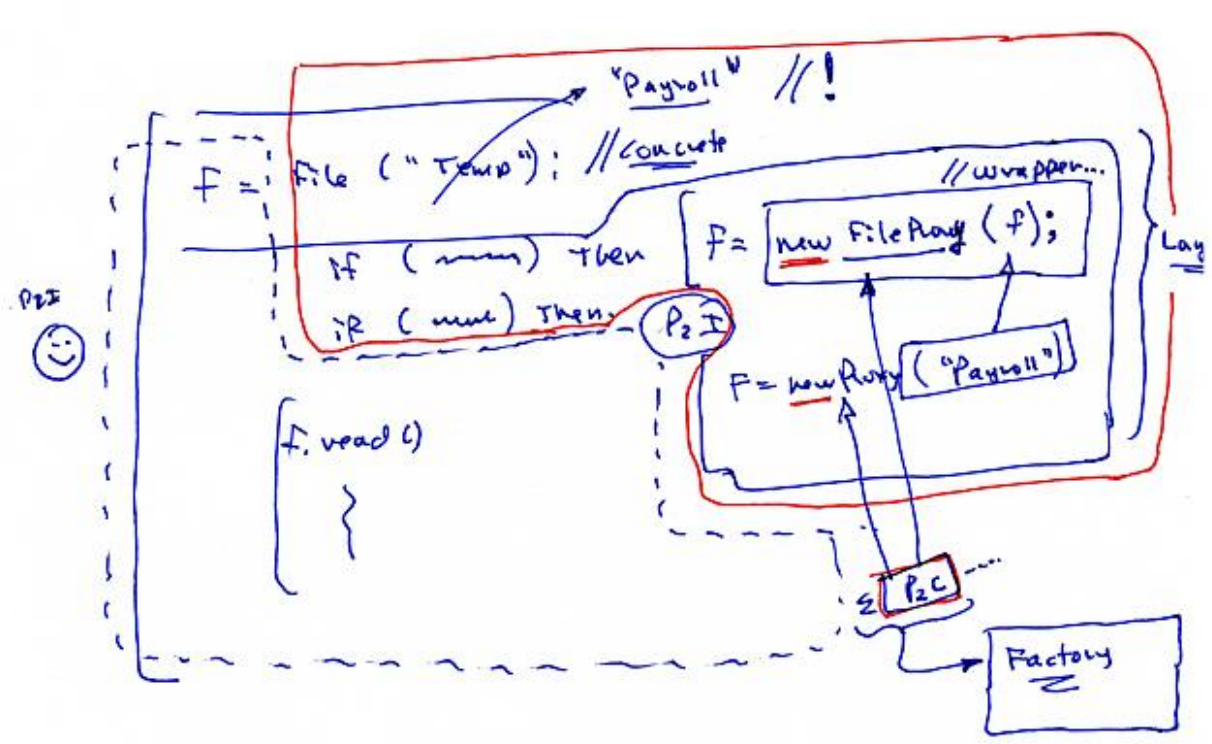
- Implementation choices for proxy wrappers; composition and inheritance
- Inheritance model is not so good…



- Composition model ⇨ Good!



- Result of using composition model for proxy

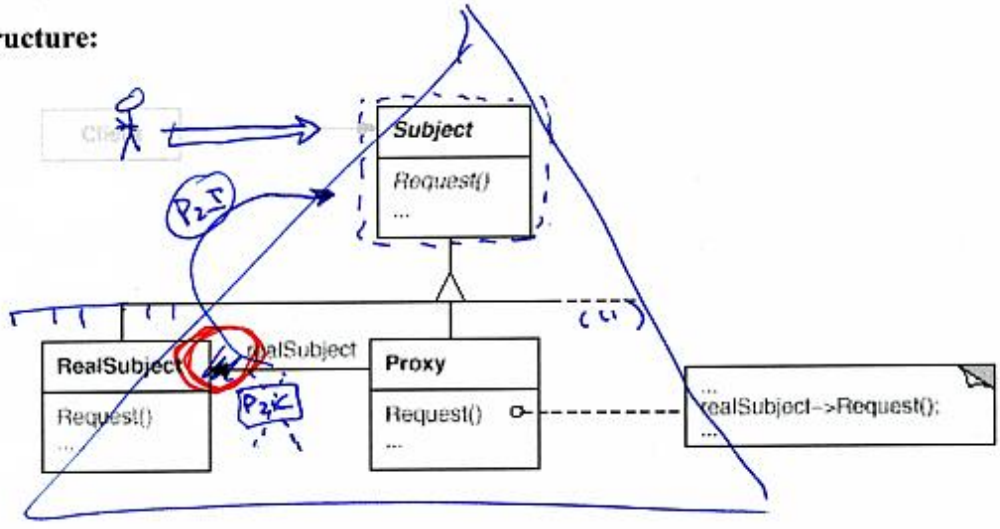


- Proxy allows transparent (P2I) substitution of wrappered (proxy) objects for "real"

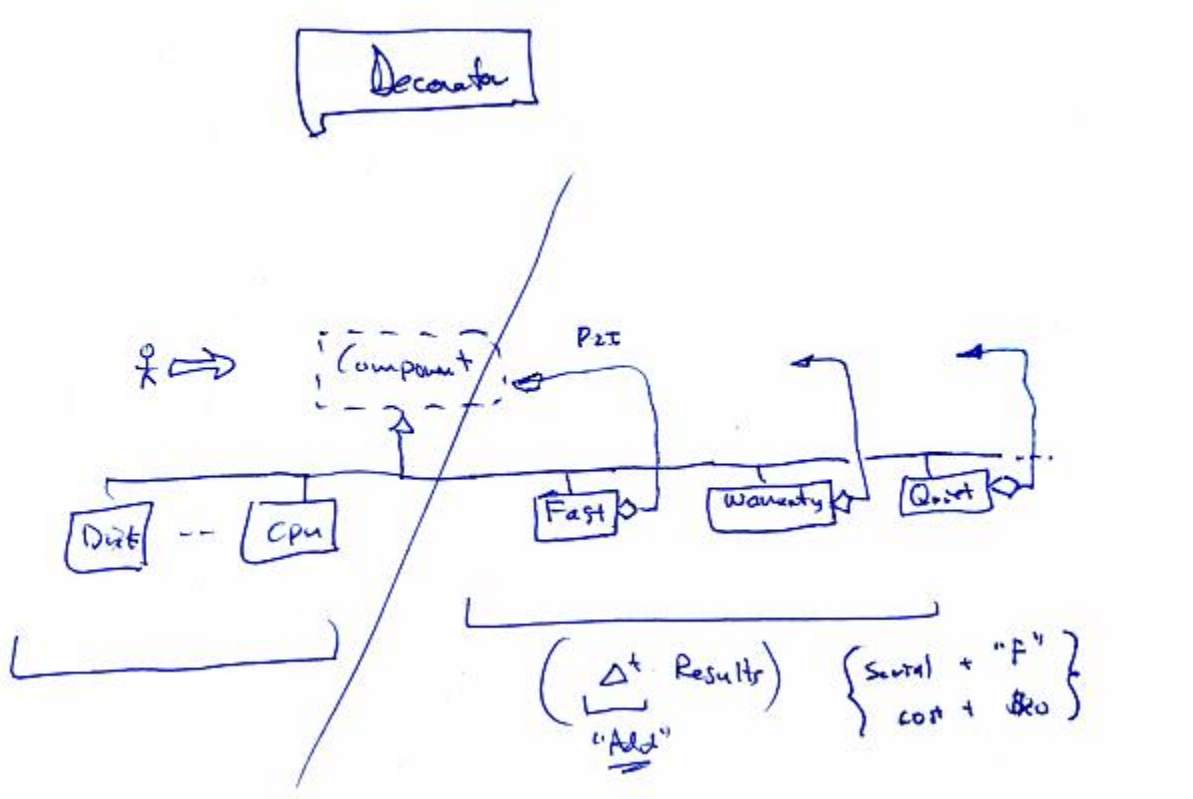
Proxy Pattern

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

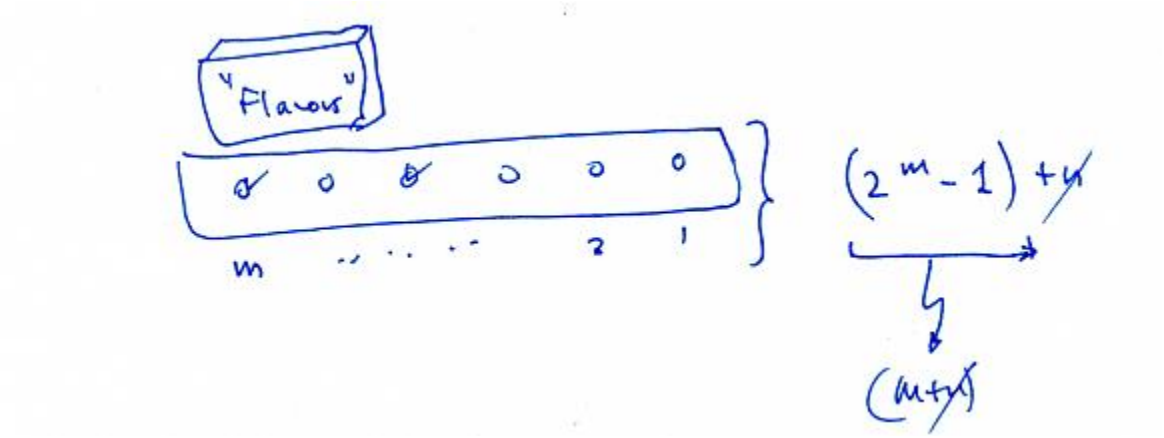
Structure:



- GOF description



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



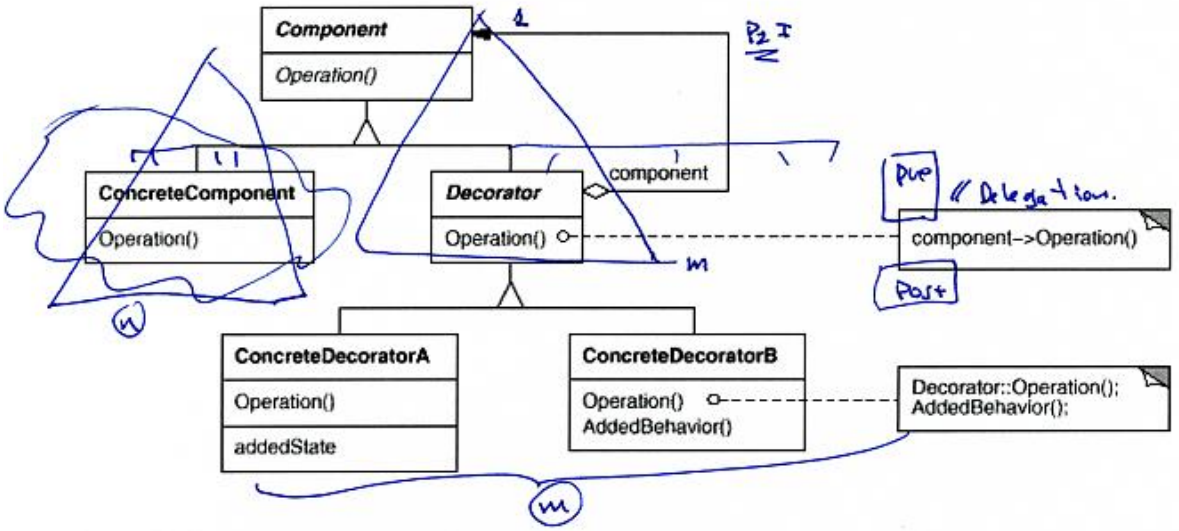
- Counting the number of static classes needed for inheritance model for proxy (or decorator)

cs525: Pattern Notes

Decorator Pattern

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

Structure:

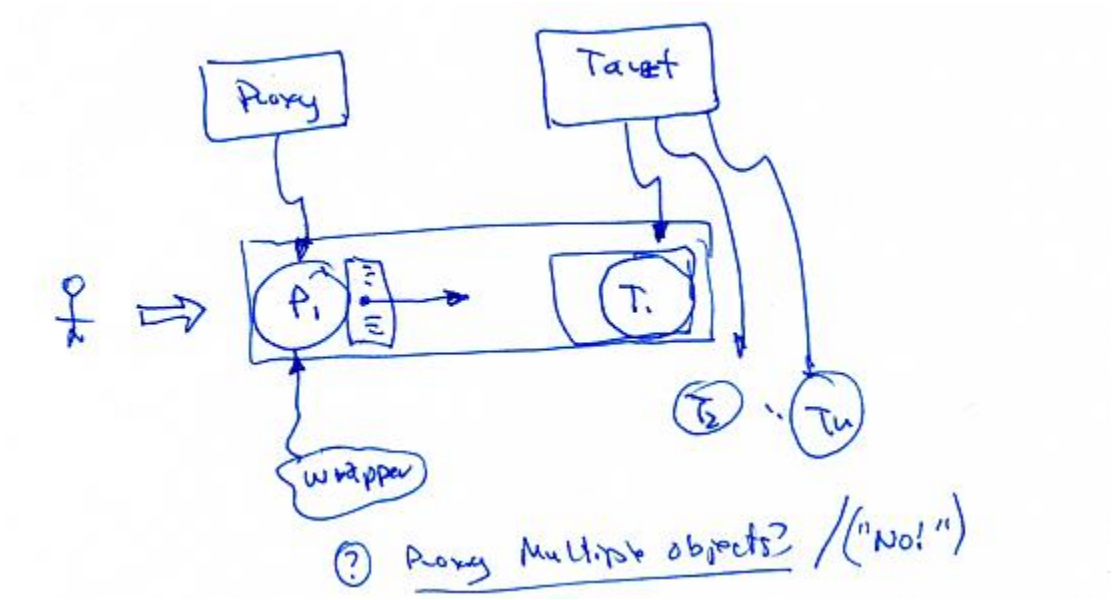


- GOF Description

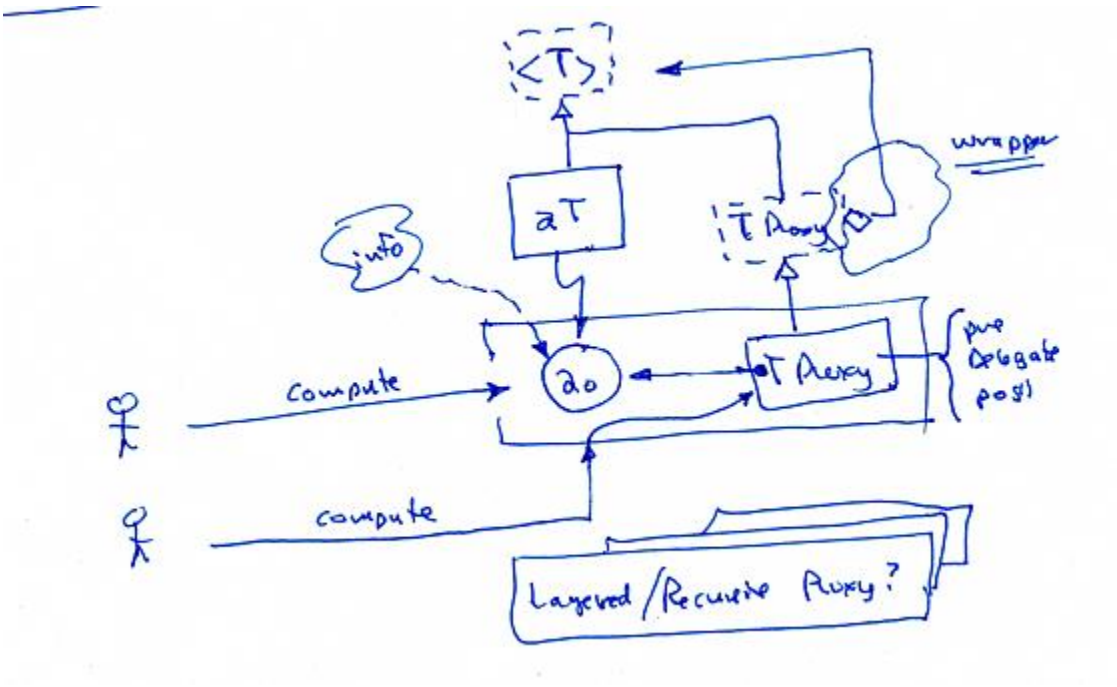
Day 12: More Proxy; Dynamic & Generic Proxy

Day 12: Proxy Fanciness, & Dynamic Proxy

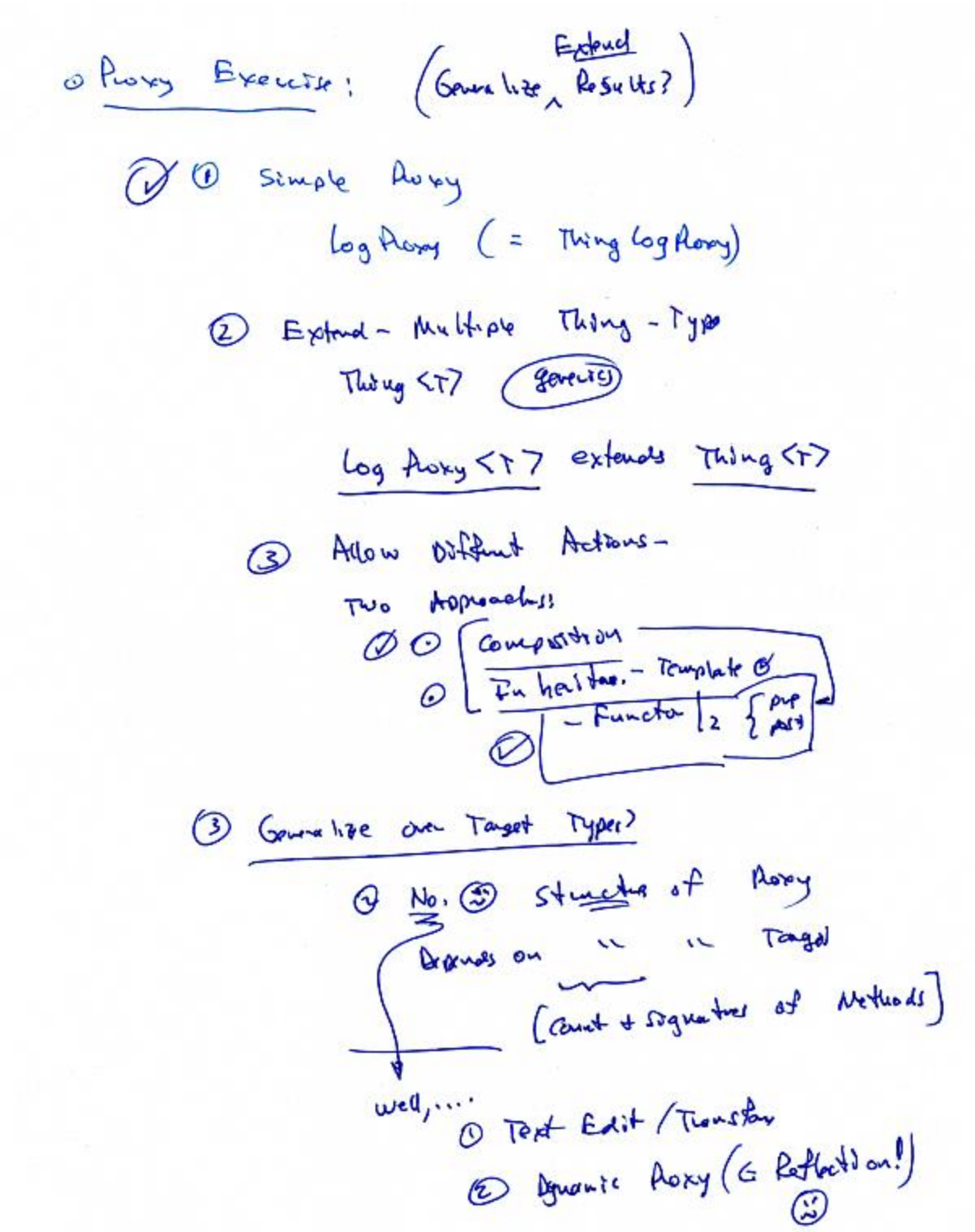
- Review *Proxy Pattern*
  - *dynamic* layers of intelligence
  - ~~n*(m+1)~~ static → (n+m) *dynamic*
- Examples:
  - Memoization proxy
- Proxy++
  - increased generality through abstractions
  - derived proxy classes
    - override wrapper methods
    - use template pattern, create *pre()* & *post()* 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



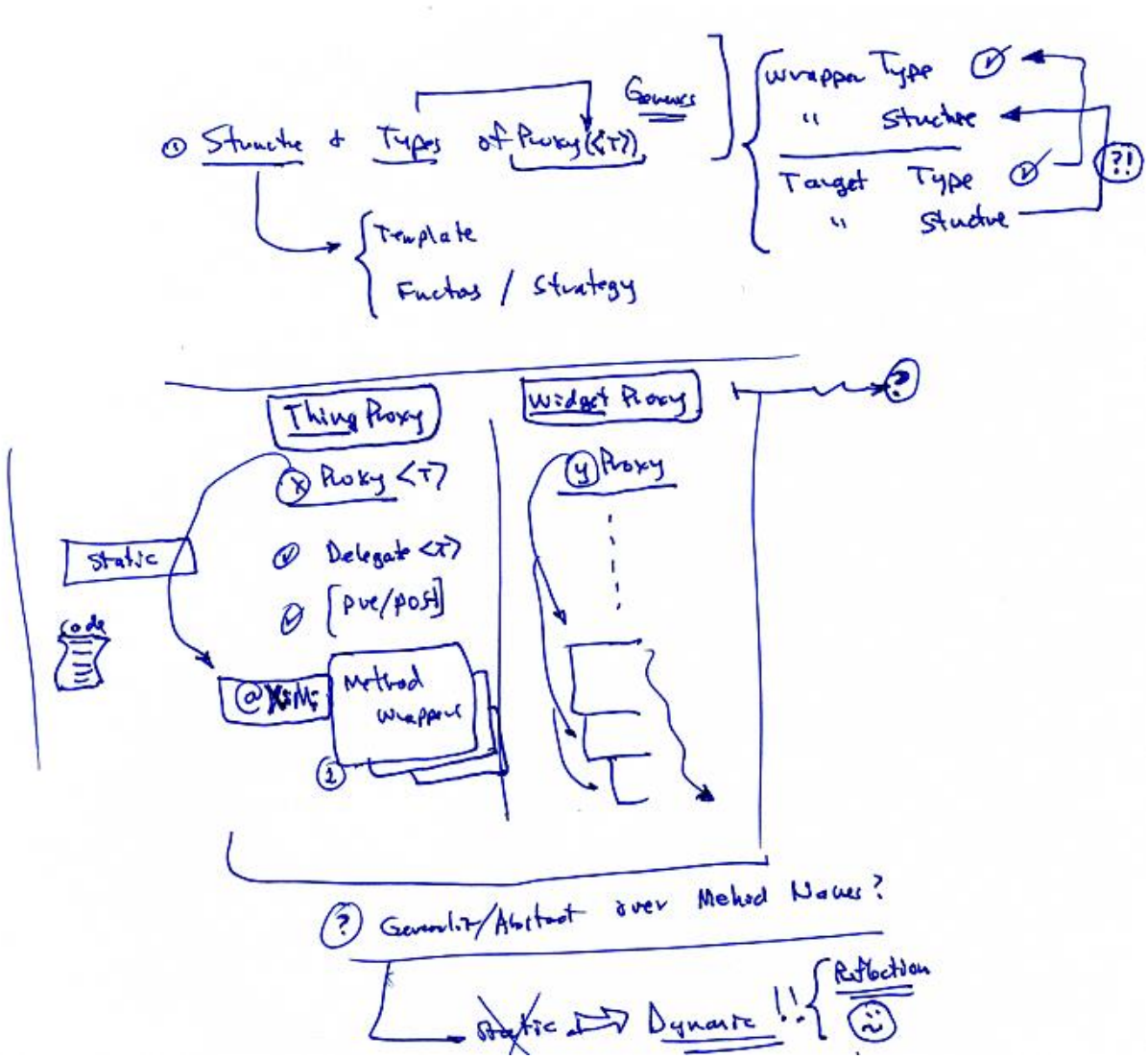
- Review question:
  Can one proxy multiple objects? (No.)



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



- Proxy exercise, and follow-up lab



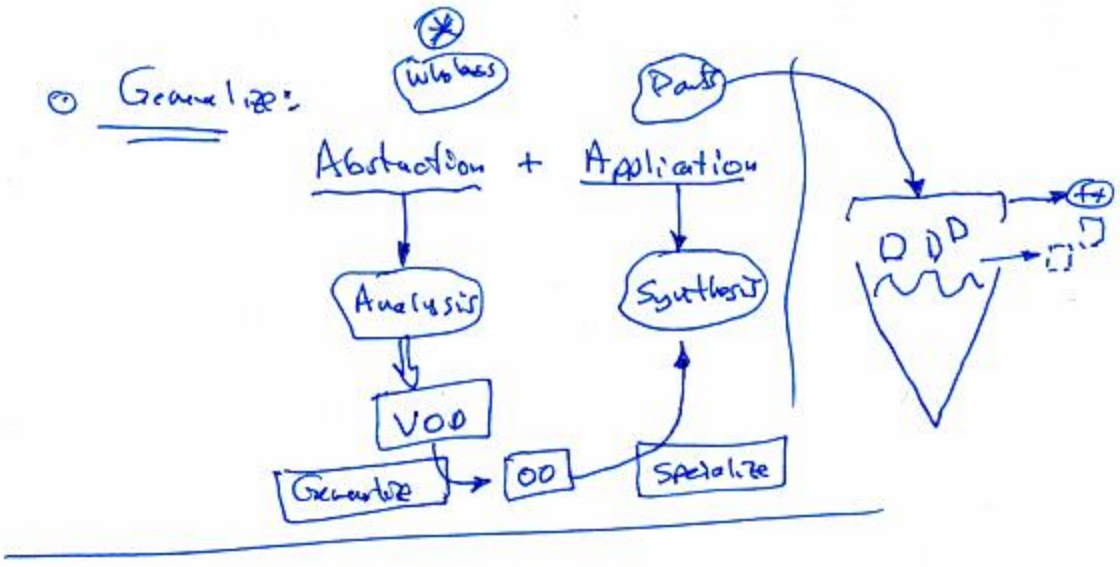
- Summary of issues with trying to generalize a proxy over target object types

Day 13: Dynamic Proxy, and Factory Pattern

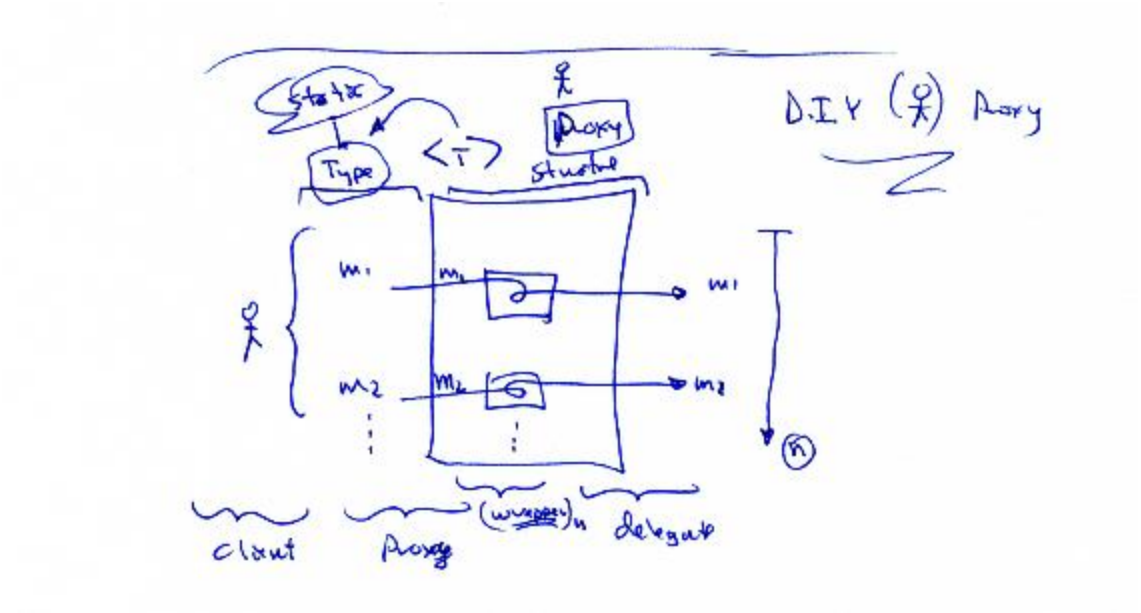
Daily Topics

Day 13: **Factory Patterns**

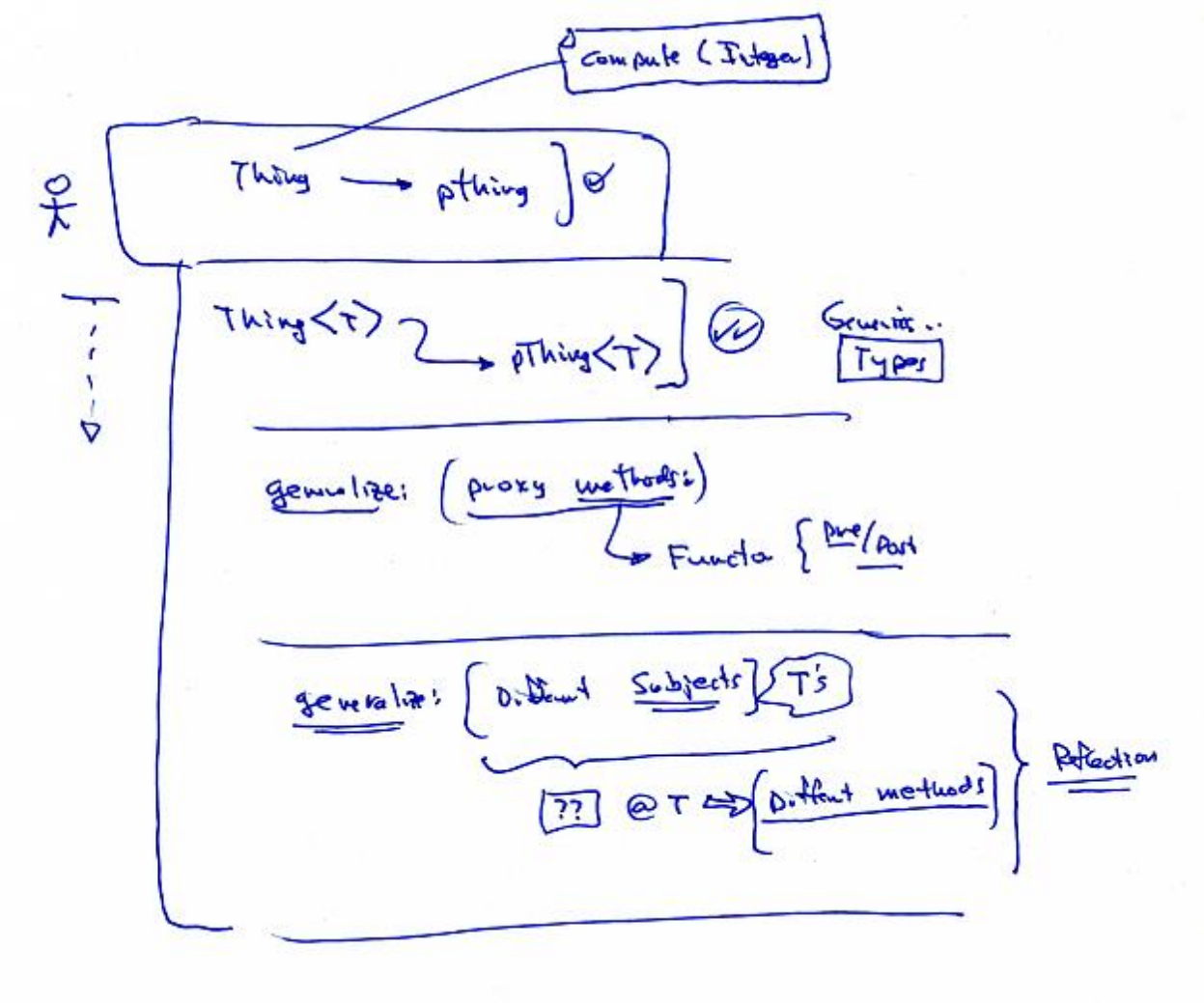
- Review ✓
- *Factory pattern* family
  - Simple Factory,
  - Factory Method,
  - Abstract Factory
  - Generic Factories?
  - Reflective Factory
- No *magic*, just OO ☺
- Singleton Pattern



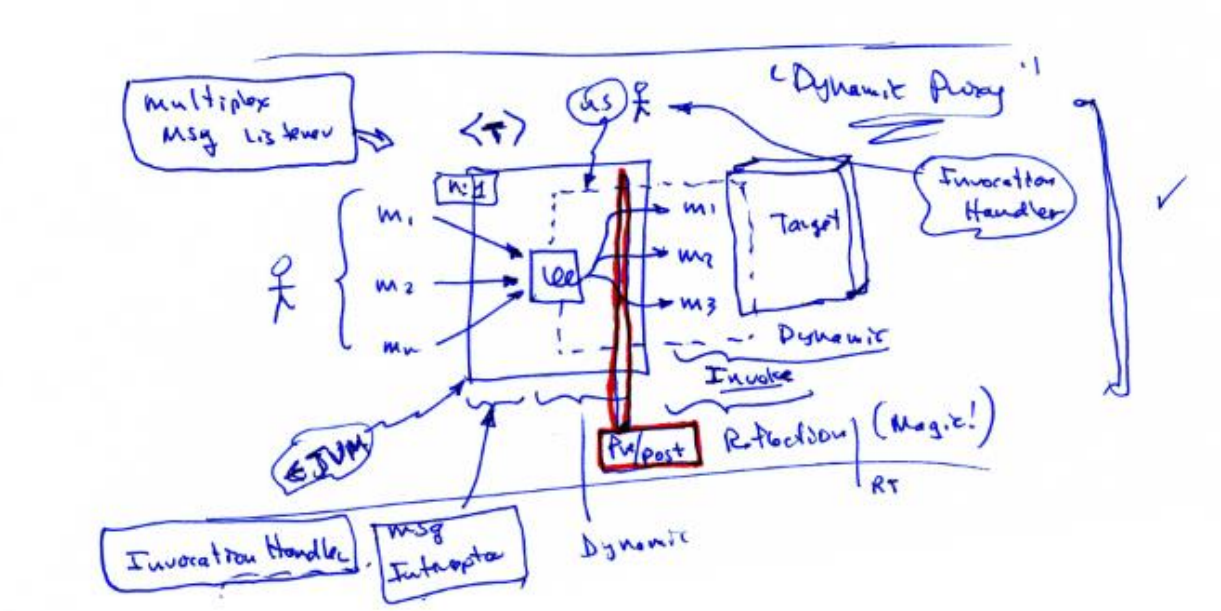
- Generalizations of a pattern and their value



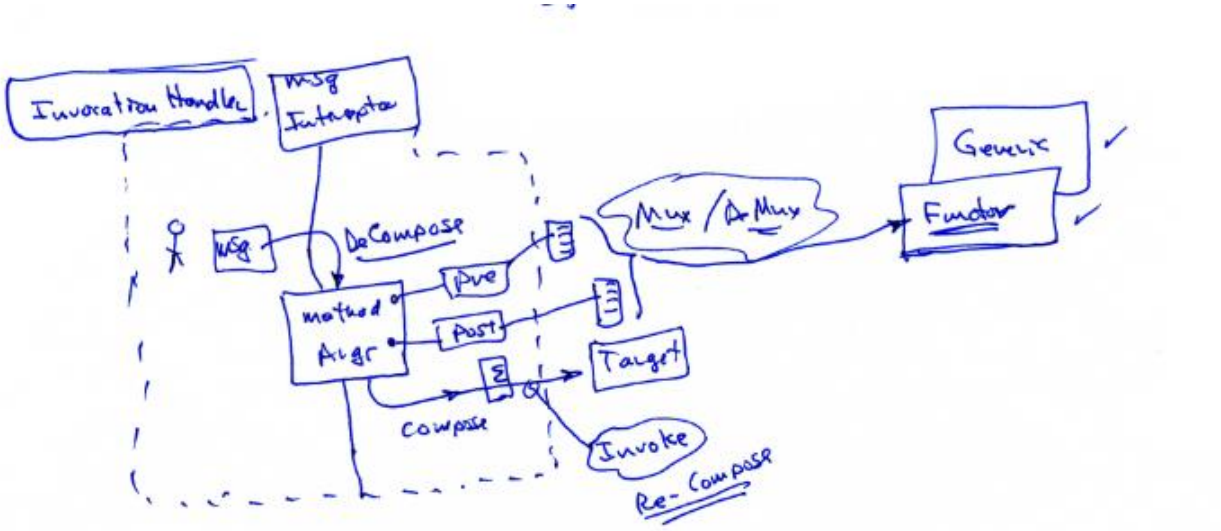
- Traditional (D.I.Y.) proxy structure, one wrapper method for every target method
- Structure of proxy ≡ structure of target



- Generalizations of the Proxy pattern

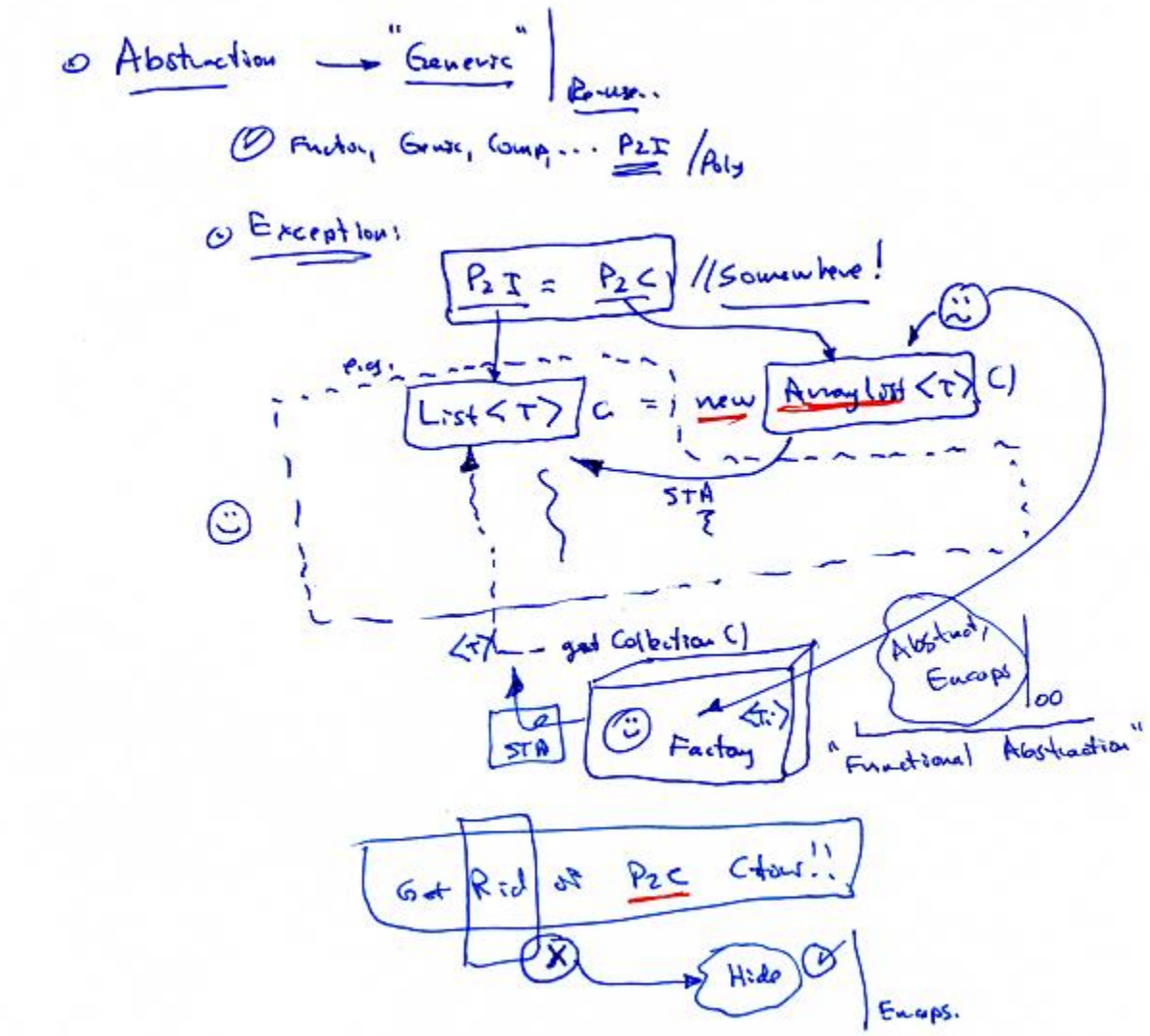


- Structure of Dynamic proxy

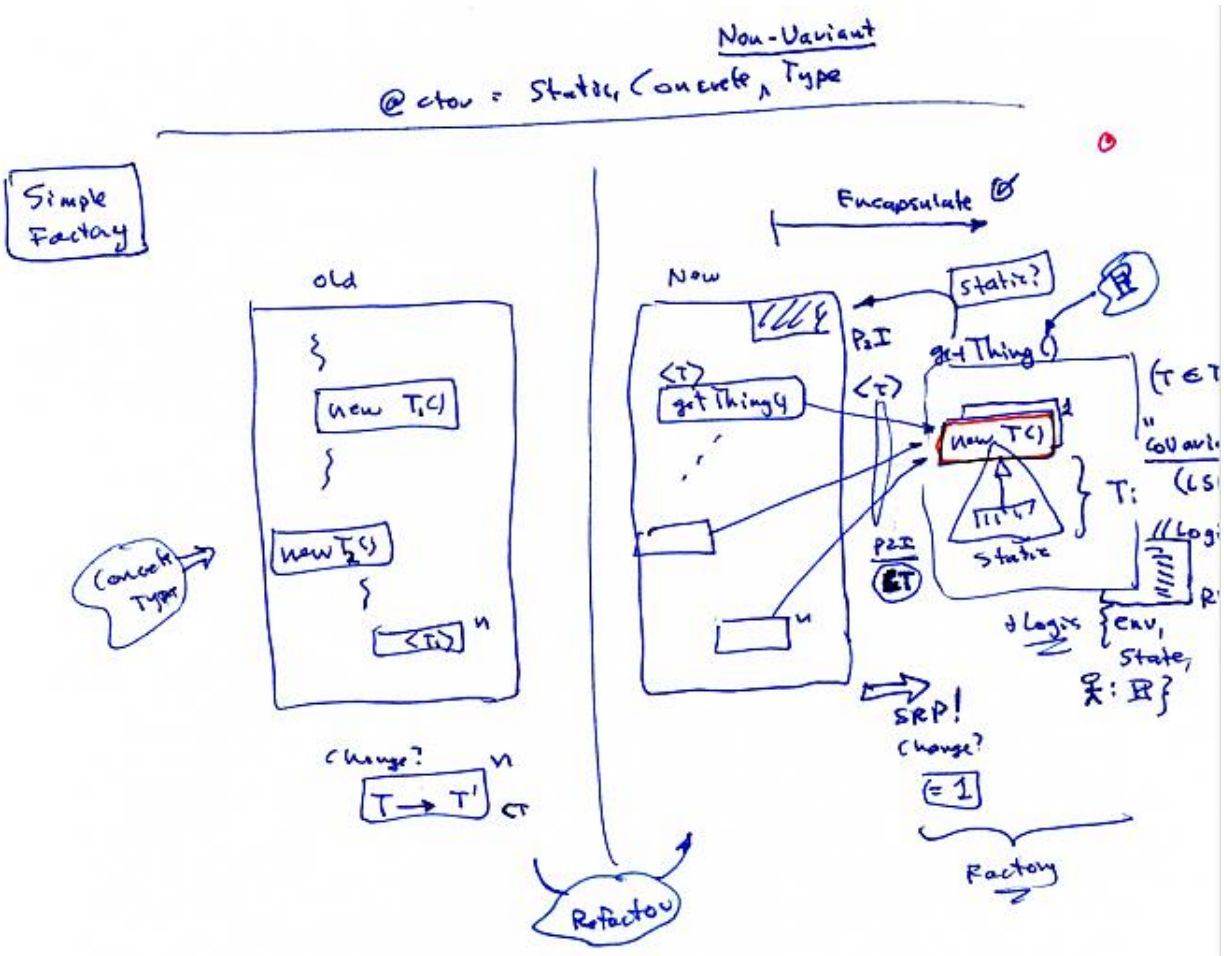


- Usage of dynamic proxy

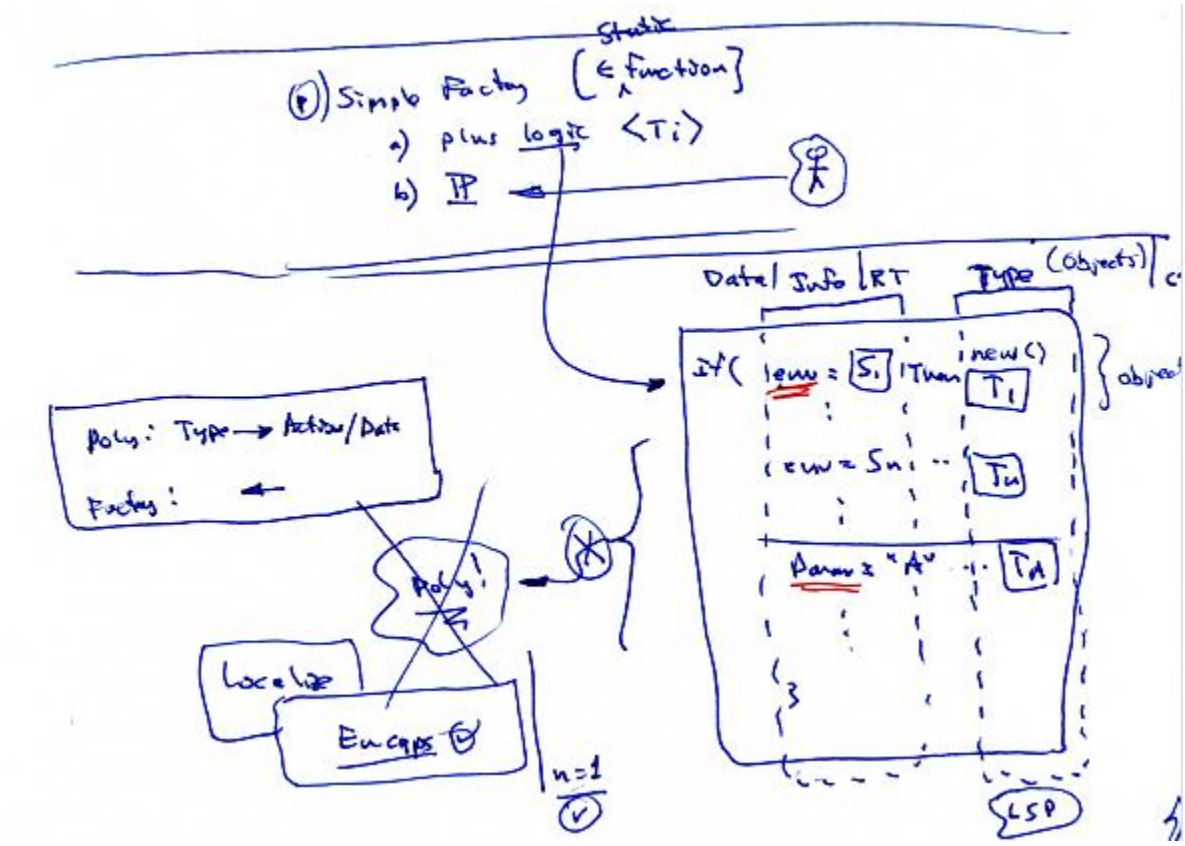
Factory Patterns ::



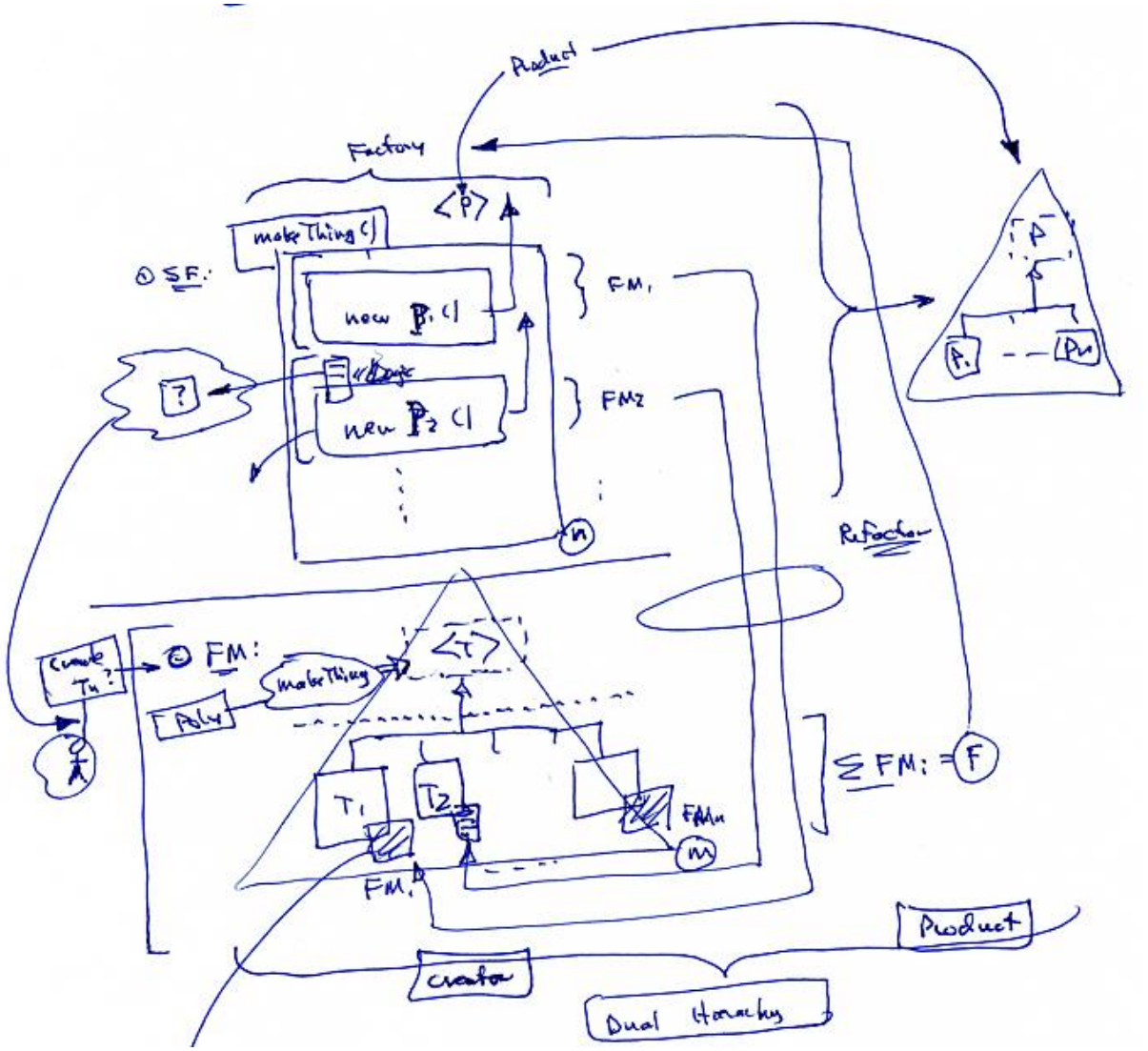
- Abstracting concrete types (*constructors*) from code



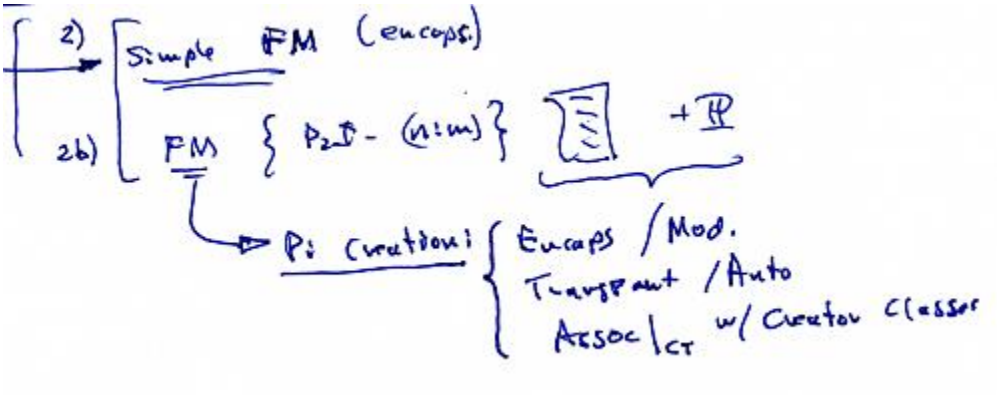
- Phase-1; Simple Factory
- Refactor concrete constructors into a single factory function



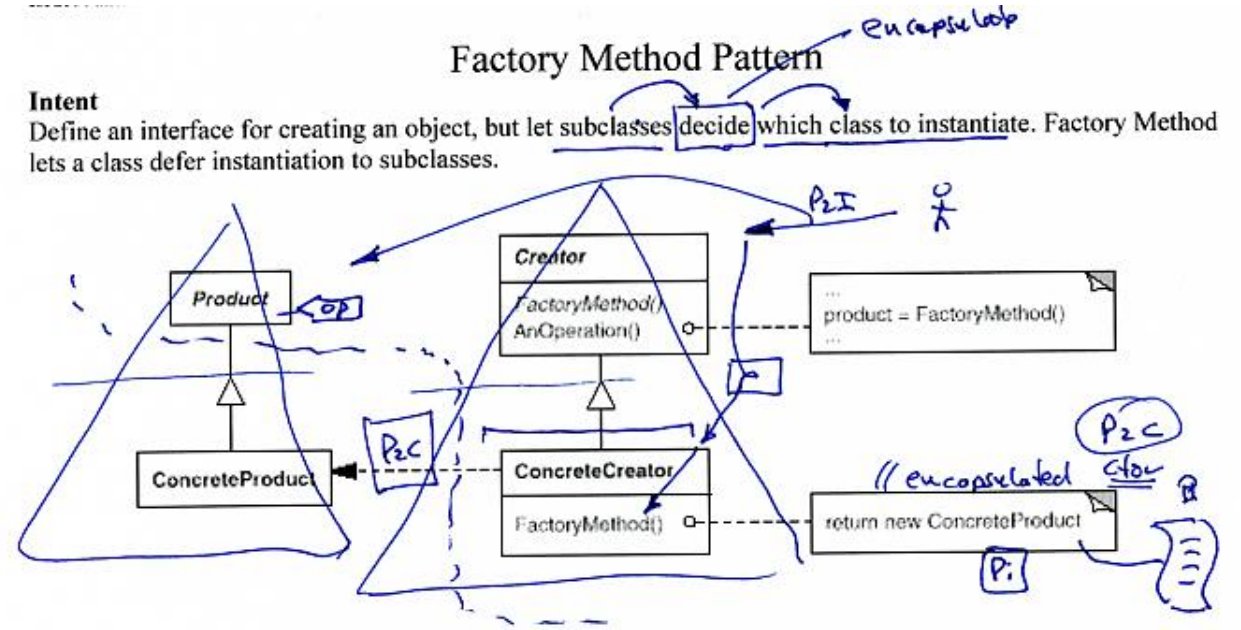
- Could also have a parameter, to allow logic for sub-type selection inside of the factory function



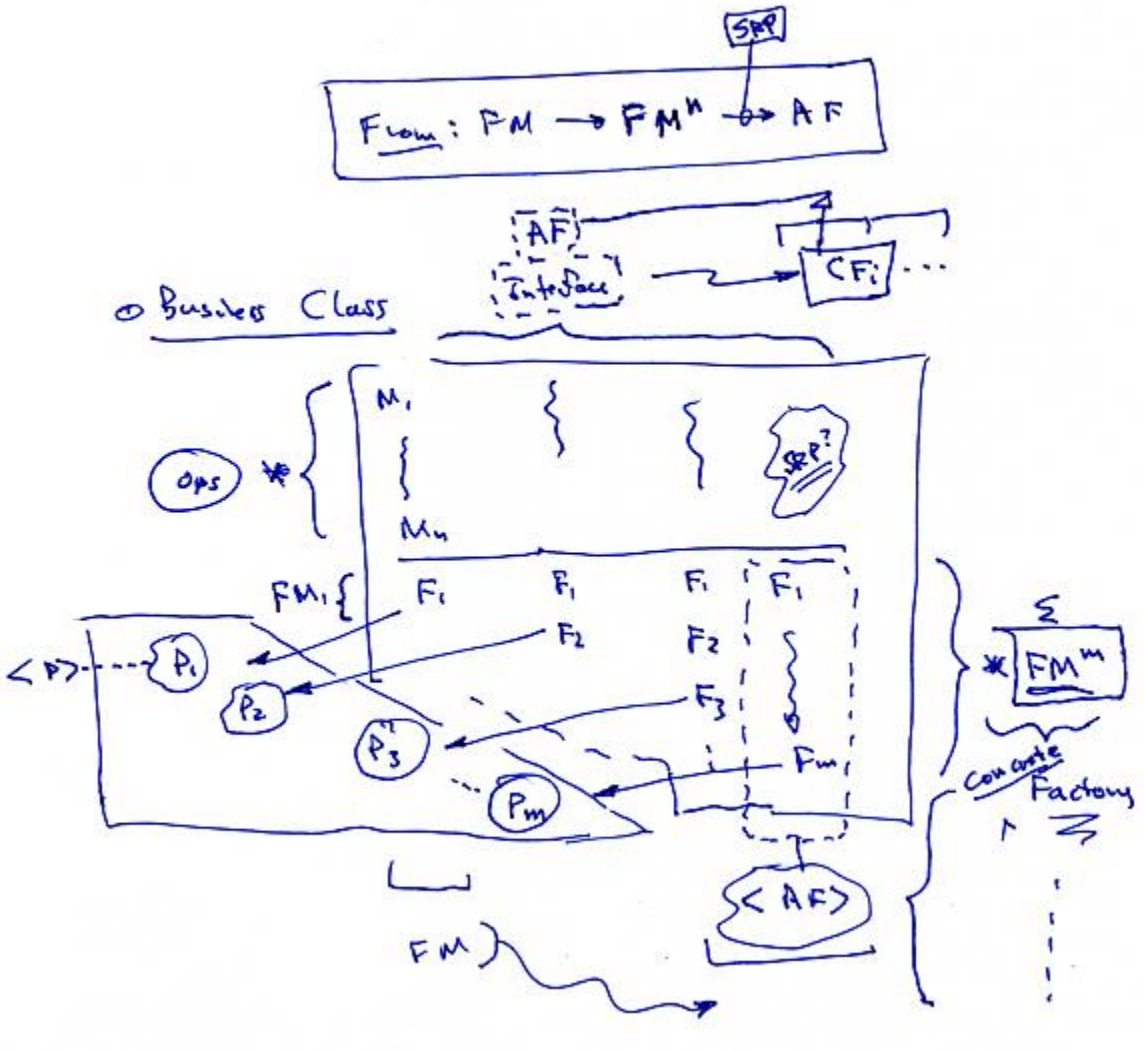
- From a Simple-Factory (function) to a factory Method;
- Refactor code for each concrete type creation into sub-classes of another hierarchy



- This is our second class of factory's



- GOF description



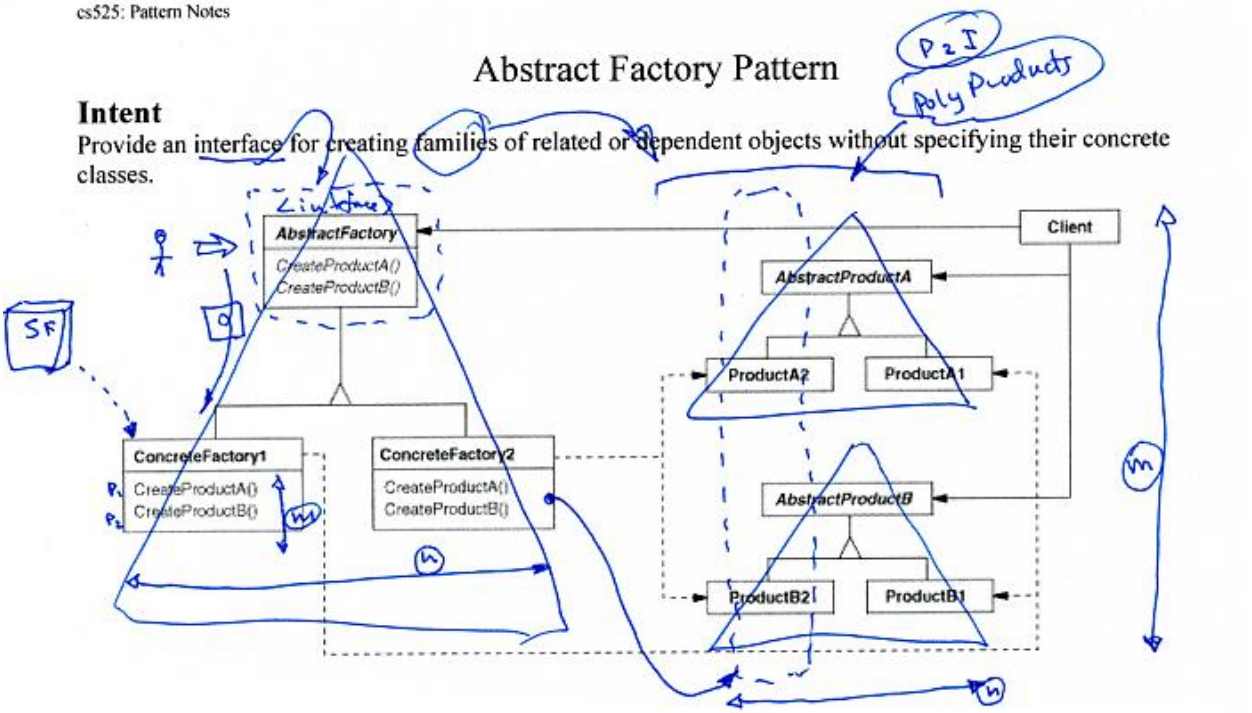
- Generalizing from FM → AF,
- AF ≈ FM^n

cs525: Pattern Notes

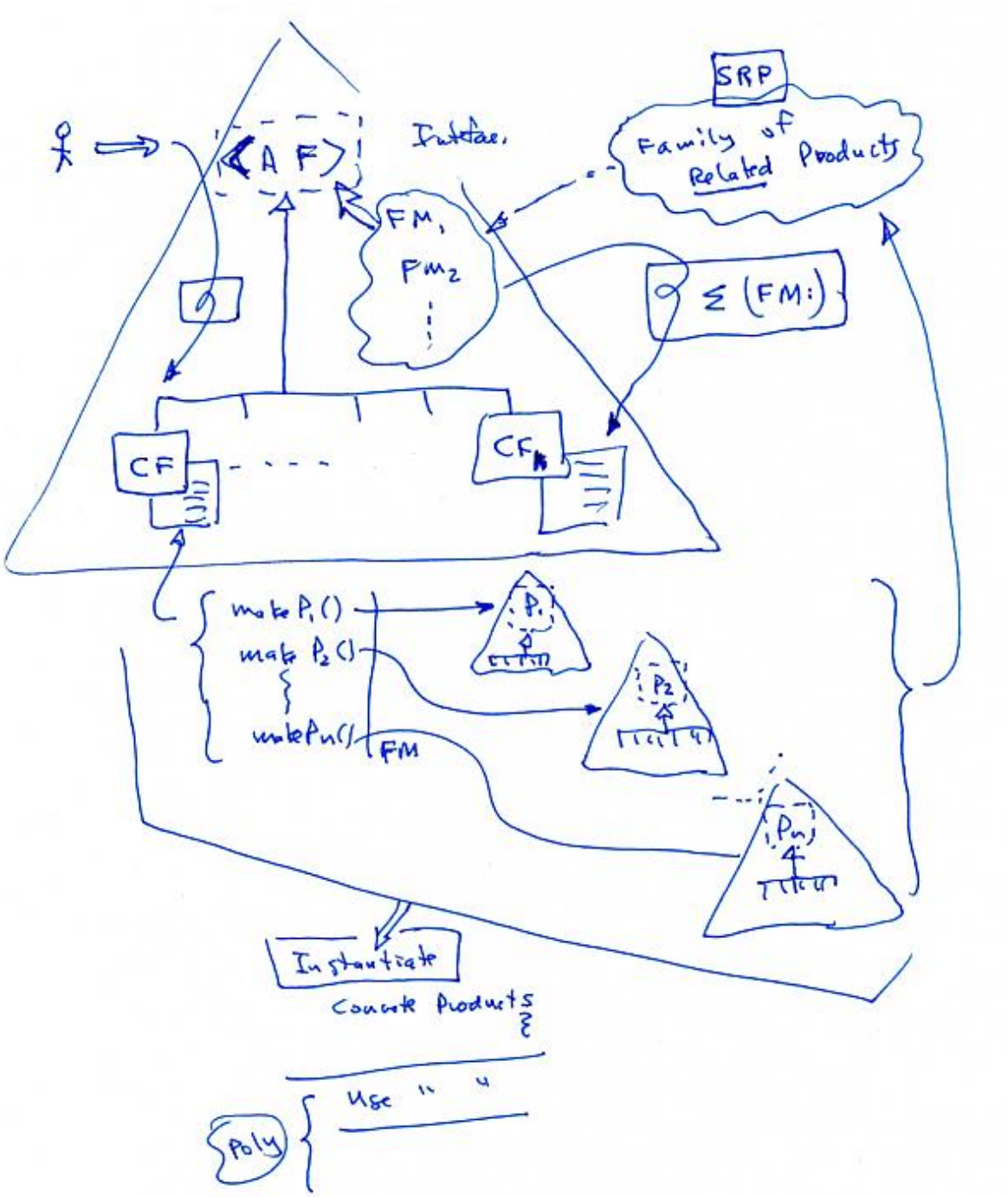
Abstract Factory Pattern

Intent

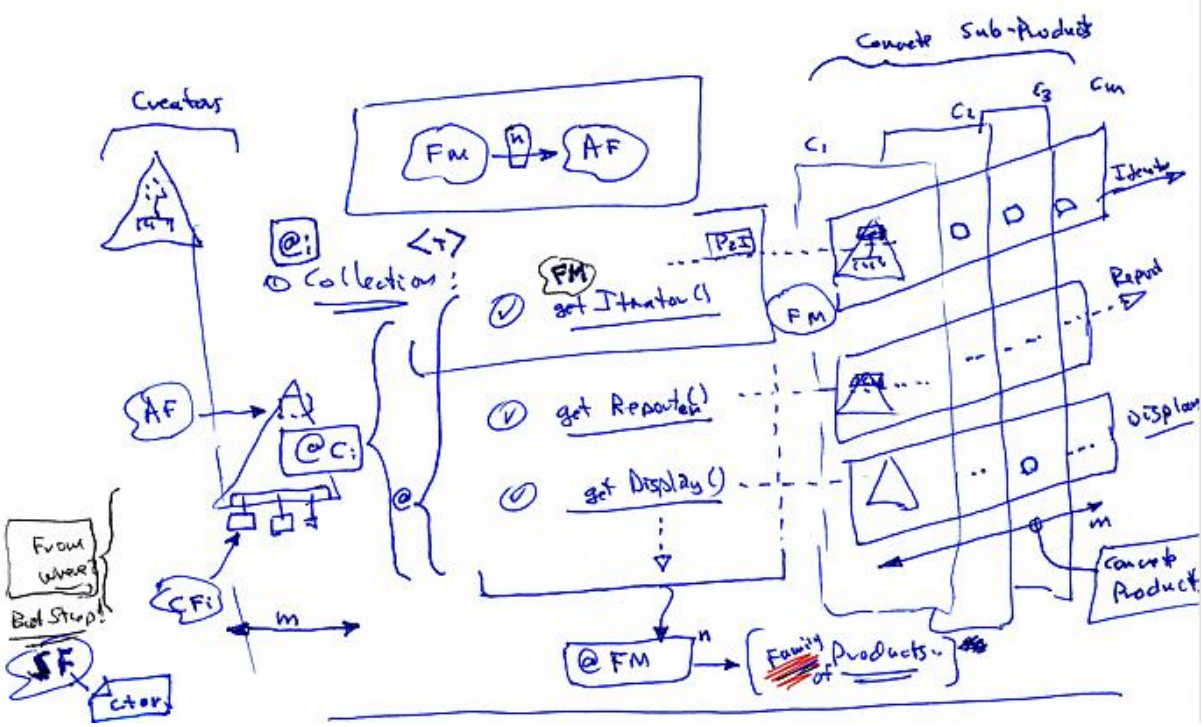
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



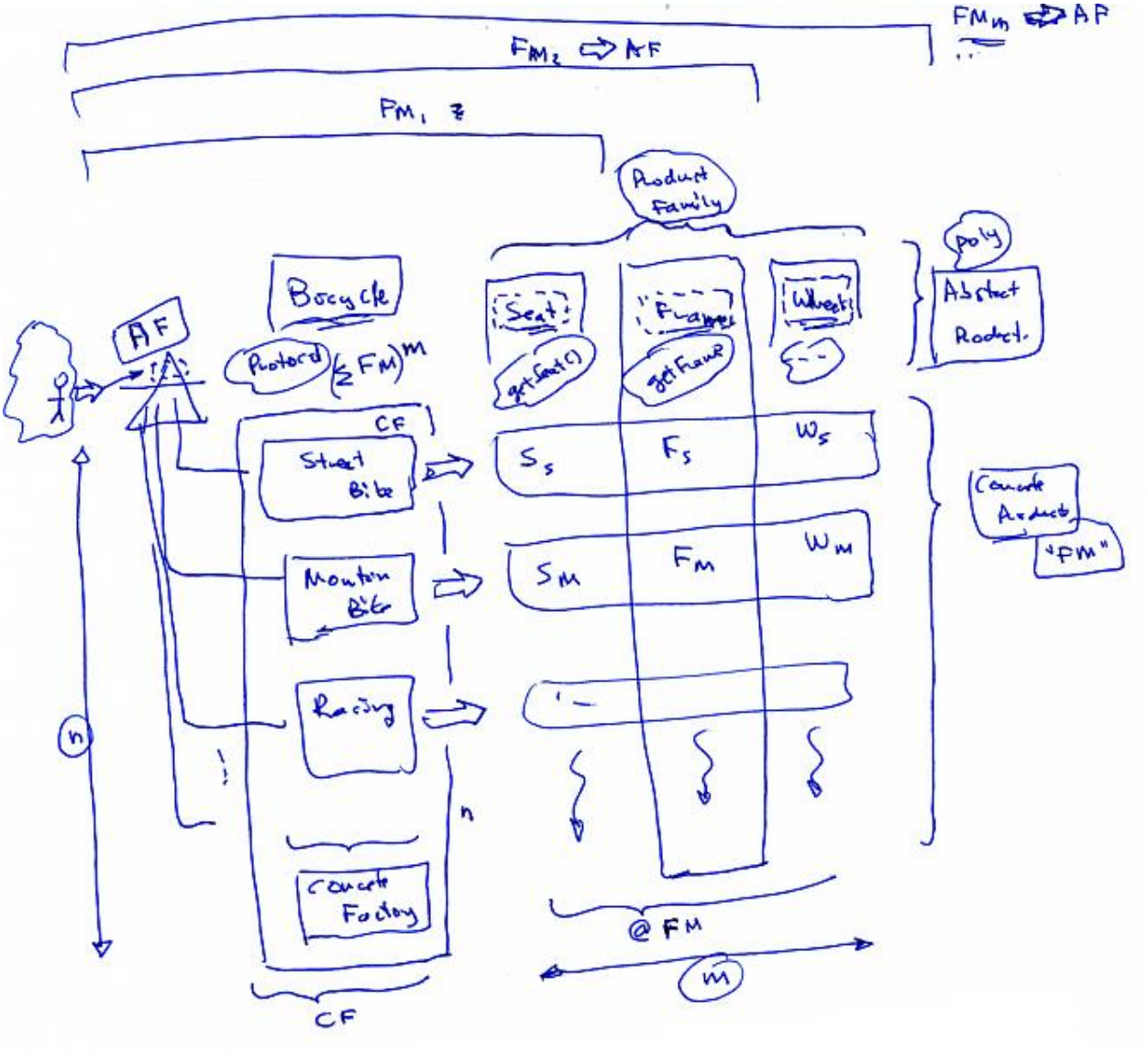
- GOF description



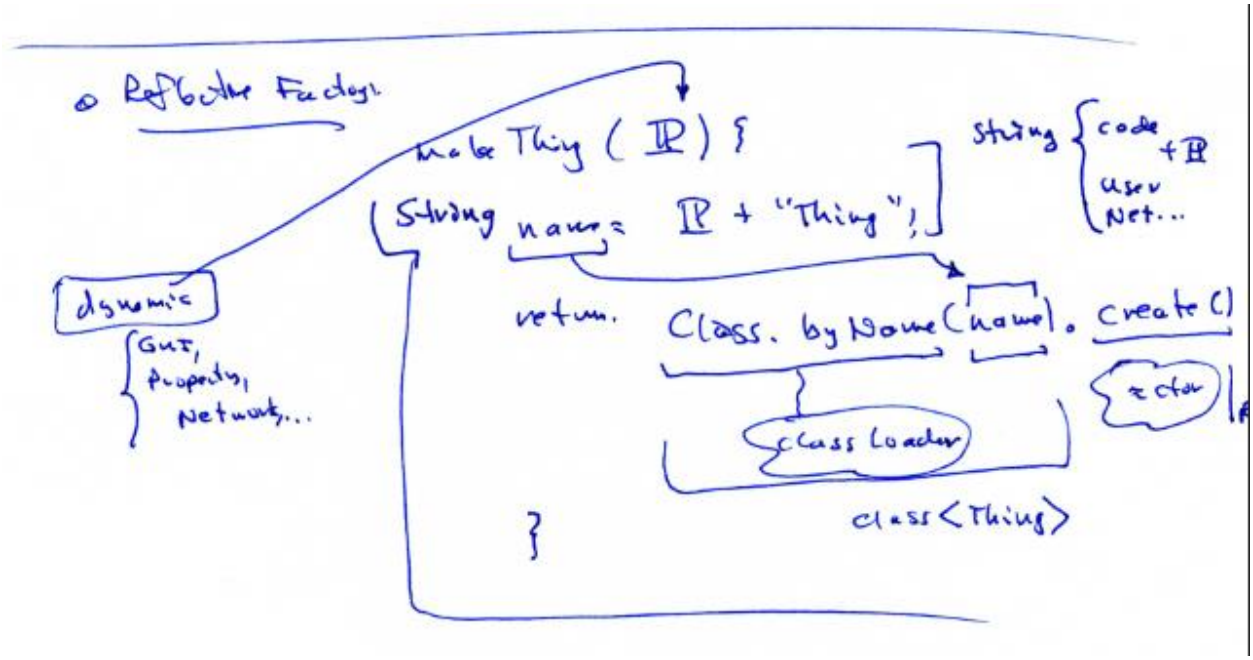
- Structure of AF pattern & usage



- Example of using AF pattern

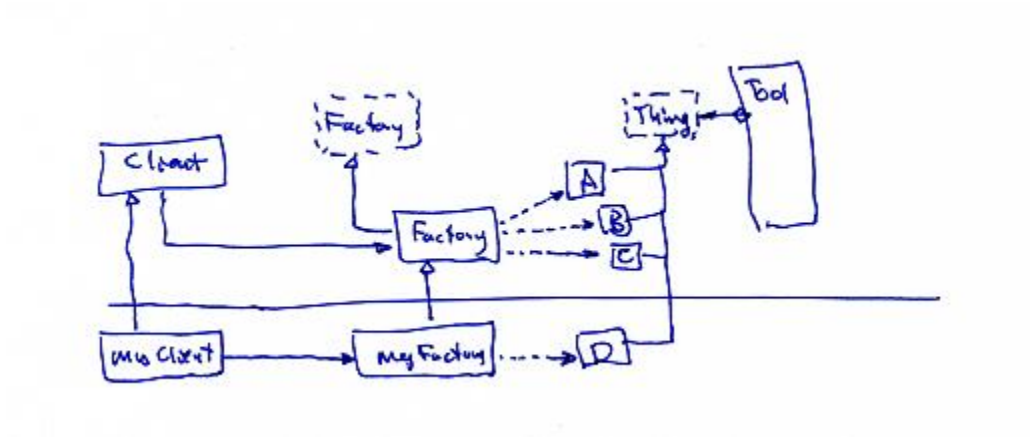


- Another AF example – bicycle shop

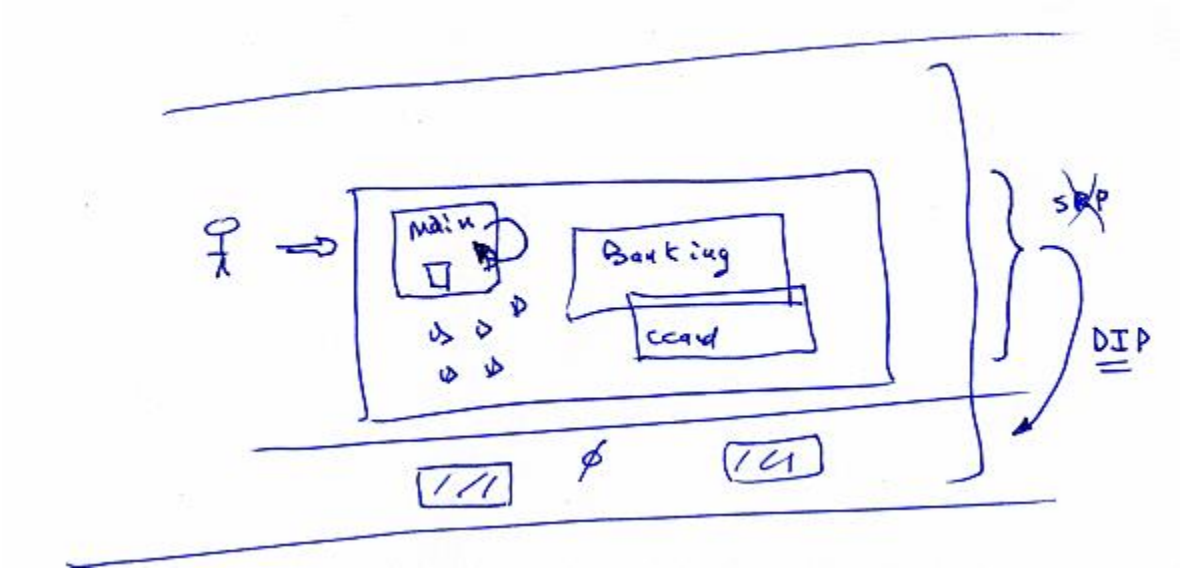


- Reflective Factory

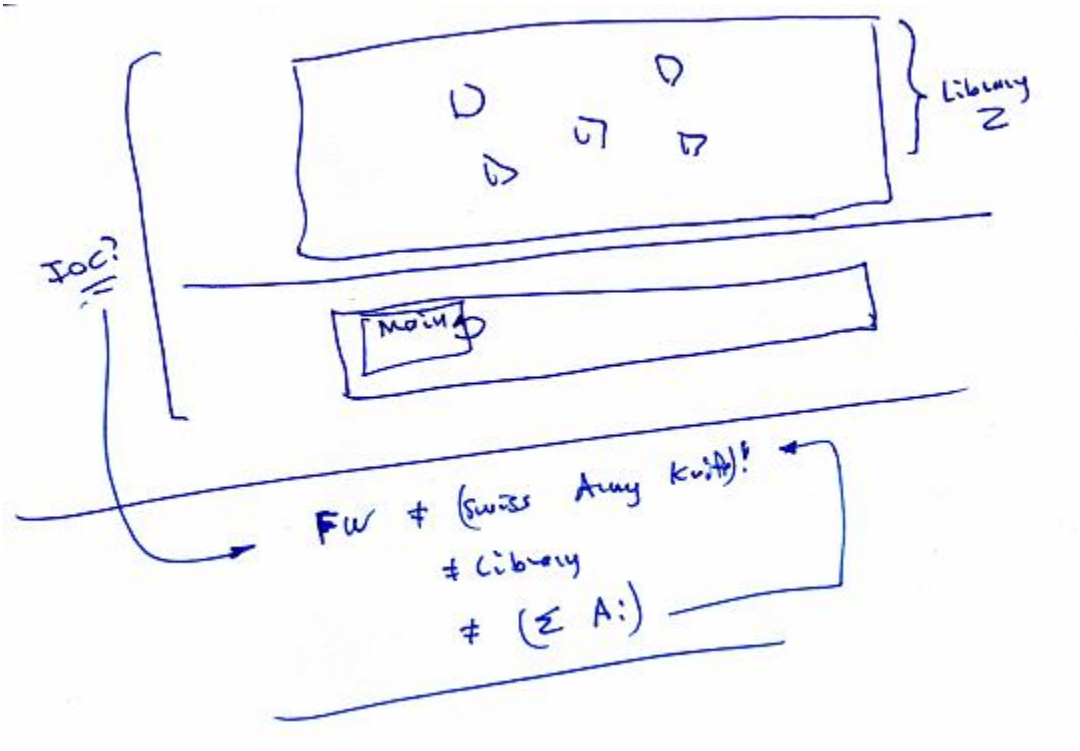
Day 14: Project Assignment



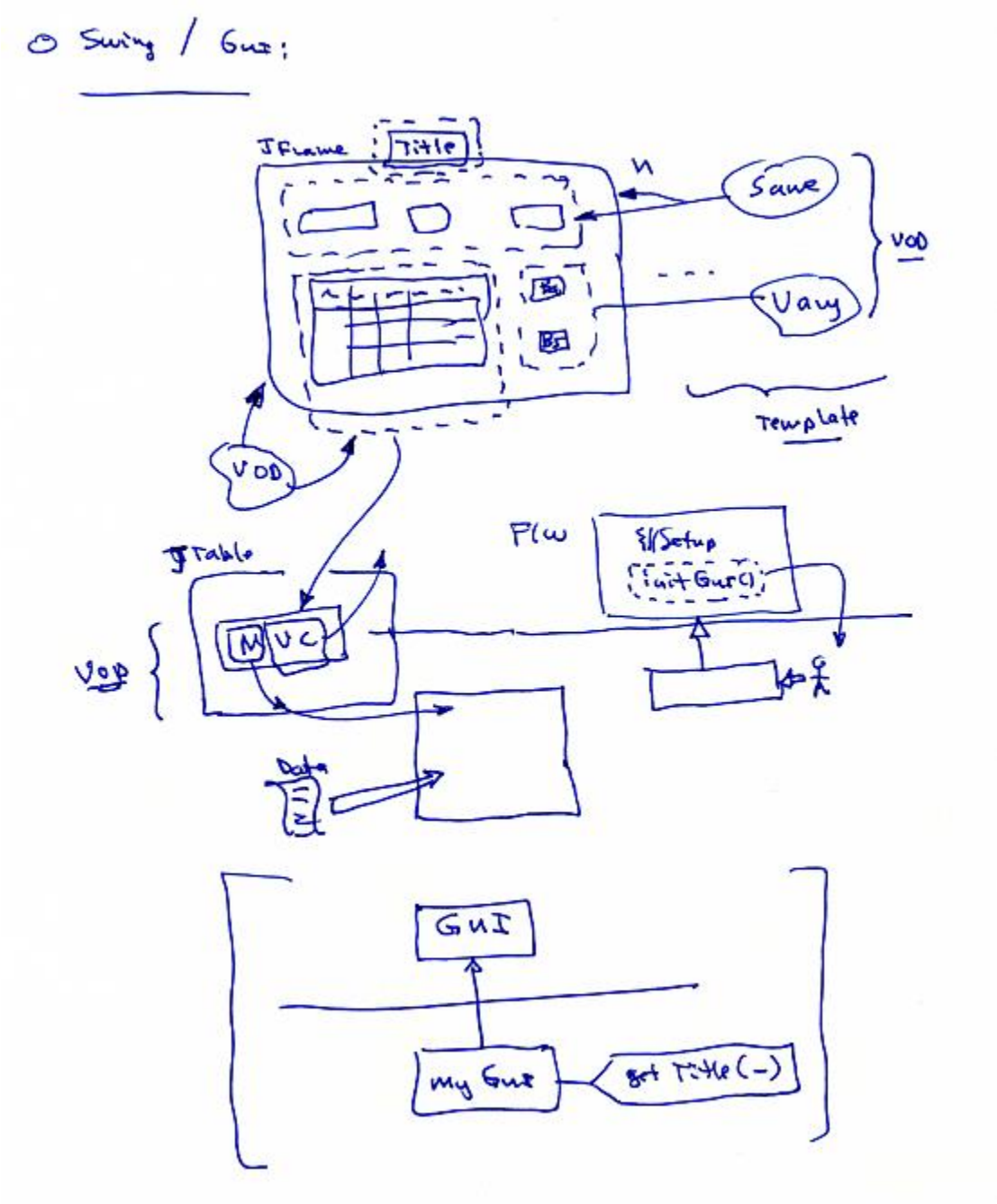
- Using factories with Frameworks



- DIP principle and frameworks



- Framework ≠ Library, or combination of applications



- Refactoring GUI for a framework