SW4, Restructuring CODE

A. First Contact

Note:
The following OORP - p.xx refers to a page in the book "Object-Oriented Reengineering Patterns". For those who are interested (not required) can find more details about the topic in the book. The book is free for download under http://scg.unibe.ch/download/oorp /)

Task 1. Skim the Documentation (OORP – p. 61)

You decide to first have a look at the documentation that comes with the system.
1. What documentation do you have?

A 4-page PDF. It holds following chapters: Use Cases, Scenario, Design / Detailed Design
There's also some detailed Javadoc in .html format and a "ToDo"-List as .txt

2. What are your first impressions about the system? Where would you focus your refactoring efforts? (Discuss with your neighbor)

The class "network" holds nearly the whole code / all methods. Maybe lay the focus on splitting of this class. All other classes serve mainly as data-holders. Obviously, there are plans to even enlarge the "network"-class, so before that, better start refactoring.

Task 2. Read all the Code in 5 Minutes (OORP – p. 53)

Next, you try to confirm some of your initial impressions by reading the source code.
1. What are your second impressions about the system? Where do you agree or disagree with your first impressions? Having a better feeling about the code, where would you now focus your refactoring efforts? (Discuss with your neighbor)

The class "network" is a so called "god-class". The class has duplicated code, nested conditionals, navigation code. I would start to split up the network class, introduce interfaces and more / better tests.

Task 3. Do a Mock Installation (OORP – p. 77)

Finally, you try to run the code and the regression tests that come along with it.
1. Try to compile and run (Eclipse/make)
2. Run the regression tests (JUnit Testrunner/make runtests)
3. Change a few lines here in the regression tests and the code to verify whether the regression tests do test the code you're looking at.
4. Have a look at the regression tests and see whether they cover all the use cases.

1. Do you feel you the code base is ready to be refactored? How about the quality of the regression tests: can you safely start to refactor? (Discuss with your neighbor)
The code base is ready to be refactored. (evtl. auch nicht, spielt eigentlich keine Rolle für die Aufgabe)

B. Reengineer

Task 4. Extract Method (OORP – p. 317)

One of the things you might have seen is that there is a considerable amount of duplicated code which represents important domain logic inside the class "Network". You decide to first get rid of some of the duplicated code (which refactoring can you apply?).

1. The accounting code occurs twice within "printDocument". Get rid of the clones.
2. The logging code occurs three times, twice inside "requestWorkstationPrintsDocument" and once inside "requestBroadcast". Get rid of the clones. Note that this time the three clones are not exactly the same so you'll first have to massage the code a bit before doing the refactoring.
3. Is there any other duplicated code representing important domain logic which should be refactored? Can you refactor it?

Are you confident that these refactorings did not break the code? Do you believe that these refactorings are worthwhile? Does the tool do a good job? (Discuss with your neighbor)

"Extract Method" -> extract the code on both places and create a new method "doAccounting"

Intent: You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method.

Task 5. Move Behavior Close to the Data (OORP – p. 243)

Having extracted the above methods, you note that none of them is referring to attributes defined on the class "Network", the class these methods are defined upon. On the other hand, these methods do access public fields from the class "Node" and "Packet".

1. The logging method you just extracted does not belong in Network because most of the data it accesses belongs in another class. Choose the appropriate refactoring to define the behavior closer to the data it operates on.
2. Similarly, the "printDocument" method is also one that accesses attributes from two faraway classes, yet does not access its own attributes.
3. Are there any other methods that are better moved closer to the data they operate on? If so, apply MOVE METHOD until you're satisfied with the results.

Are you confident that these refactorings did not break the code? Do you believe that these refactorings are worthwhile? Does the tool do a good job? (Discuss with your neighbor)

***

Intent: A method is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

Solution: Move behavior defined by indirect clients to the container of the data on which it operates.

***

Task 6. Eliminate Navigation Code (OORP – p. 253)

There is still a piece of duplicated logic left in the code, namely the way we follow the "nextNode_" pointers until we cycled through the network; logic which is duplicated both in

"requestWorkstationPrintsDocument" and "requestBroadcast" (and to a lesser degree in "printOn", "printHTMLOn", "printXMLOn"). This duplicated logic is quite vulnerable, because it accesses attributes defined on another class and in fact it represents a special kind of navigation code.
4. Apply the appropriate refactoring on the boolean expression defining the end of the loop, creating a predicate "atDestination".
5. Rewrite the loops driven by the "currentNode = currentNode.nextNode_" into a recursive call of a "send" method.
Are you confident that these refactorings did not break the code? Do you believe that these refactorings are worthwhile? Does the tool do a good job? (Discuss with your neighbor)

***
Reduce the impact of changes by shifting responsibility down a chain of connected classes.

Solution: Iteratively move behavior defined by an indirect client to the container of the data on which it operates.Note that actual reengineering steps are basically the same as those
of Move Behavior Close to Data, but the manifestation of the problem is rather different, so different detection steps apply.

***
Task 7. Transform Self Type Checks (OORP – p. 273)
Another striking piece of duplicated logic can be found in "printOn", "printHTMLOn", "printXMLOn". However, this time it is a duplicated conditional, and given the extra functionality (namely the introduction of a GATEWAY node) one that is likely to change. Thus it is worthwhile to introduce new subclasses here.
6. Normally, you should have noticed during MOVE BEHAVIOUR CLOSE TO DATA change, that the switch statements inside "printOn", "printHTMLOn", "printXMLOn" should have been extracted and moved onto the class Node. If you didn't, do that now, naming the new methods "printOn", "printHTMLOn", "printXMLOn".

7. Create empty subclasses for the different types of Node that do exist ("WorkStation", "Printer").
8. Patch the constructor clients of Node so that they know create instances of the appropriate class.
9. Move the code from the legs of the conditional into the appropriate (sub)class, eventually removing the conditional.
10. Verify all accesses to the "type_" attribute of Node. As long as you'll find any keep doing a Transform Self Type Checks or Transform Client Type Checks until you completely removed them all.
11. Remove the "type_" attribute.

Are you confident that these refactorings did not break the code? Do you believe that these refactorings are worthwhile? Does the tool do a good job? (Discuss with your neighbor)

***
Intent: Improve the extensibility of a class by replacing a complex conditional statement with a call to a hook method implemented by subclasses.

Solution: Identify the methods with complex conditional branches. In each case, replace the conditional code with a call to a new hook method. Identify or introduce subclasses corresponding to the cases of the conditional. In each of these subclasses, implement the hook method with the code corresponding to that case in the original case statement.
***

Software Architecture: Unit 1

Exercise 1

Please read the article “Evolutionary architecture and emergent design: Investigating architecture and design” by Neal Ford (see http://www.ibm.com/developerworks/java/library/j-eaed1/index.html) and have our definitions for software architecture at hand. Try to answer the following questions.

- Is the decision using a particular web framework an architectural one?
- Is the decision using a particular programming language an architectural one?

Answer the above questions relative to:

- the definition of Taylor et al.
- the definition of Bass et al.
- Martin Fowler's definition
- Neal Ford's definition.

Exercise 2

The following is an item taken from „97 Things Every Software Architect Should Know“, by Richard Monson-Haefel (editor), O'Reilly, 2009:

> Use uncertainty as a driver
>
> Confronted with two options, most people think that the most important thing to do is to make a choice between them. In design (software or otherwise), it is not. The presence of two options is an indicator that you need to consider uncertainty in the design. Use the uncertainty as a driver to determine where you can defer commitment to details and where you can partition and abstract to reduce the significance of design decisions. If you hardwire the first thing that comes to mind you're more likely to be stuck with it, so that incidental decisions become significant and the softness of the software is reduced.
>
> One of the simplest and most constructive definitions of architecture comes from Grady Booch: “All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.” What follows from this is that an effective architecture is one that generally reduces the significance of design decisions. An ineffective architecture will amplify significance.
>
> When a design decision can reasonably go one of two ways, an architect needs to take a step back. Instead of trying to decide between options A and B, the question becomes “How do I design so that the choice between A and B is less significant?” The most interesting thing is not actually the choice between A and B, but the fact that there is a choice between A and B (and that the appropriate choice is not necessarily obvious or stable).
>
> An architect may need to go in circles before becoming dizzy and recognizing the dichotomy. Standing at whiteboard (energetically) debating options with a colleague? Umming and ahhing in front of some code, deadlocked over whether to try one implementation or another? When a new requirement or a clarification of a requirement has cast doubt on the wisdom of a current implementation, that's uncertainty. Respond by figuring out what separation or encapsulation would isolate that decision from the code that ultimately depends on it. Without this sensibility the alternative response is often rambling code that, like a nervous interviewee, babbles away trying to compensate for uncertainty with a multitude of speculative and general options. Or, where a response is made with arbitrary but unjustified confidence, a wrong turn is taken at speed and without looking back.
>
> There is often pressure to make a decision for decision's sake. This is where options thinking can help. Where there is uncertainty over different paths a system's development might take, make the decision not to make a decision. Defer the actual decision until a decision can be made more responsibly, based on actual knowledge, but not so late that it is not possible to take advantage of the knowledge.
>
> Architecture and process are interwoven, which is a key reason that architects should favor development

> lifecycles and architectural approaches that are empirical and elicit feedback, using uncertainty constructively to divide up both the system and the schedule.
>
> By Kevlin Henney
>
>
>
> From: http://softarch.97things.oreilly.com/wiki/index.php/Use_uncertainty_as_a_driver

Tasks

Read the above text. Then preferably in groups of two persons, try to answer the following questions:

1. Given an option, what is the advice of Kevlin?

2. Given an uncertainty, what kind of means do you know that allow you to defer the corresponding decision?

Exercise 3 / Homework

The „garage door opener" example will be used in a forthcoming exercise for deriving a software architecture. Let's start investigating the general requirements:

> The controller for a garage door opener is an embedded real-time system that reacts to "open" and "close" commands from several buttons installed in the house and from a remote control unit, usually located in a car. The controller then controls the speed and direction of the motor, which opens and closes the garage door. The controller also reacts to signals from several sensors attached to the garage door. One of those sensors detects resistance to the door's movement. If the amount of resistance measured by this sensor is above a certain limit, the controller interprets that resistance as an obstacle between the garage door and the floor. As a reaction, the motor closing the garage door is stopped.
>
> From: „Deriving Architectural Tactics: A Step Toward Methodical Architectural Design" by Felix Bachmann, Len Bass, and Mark Klein; TECHNICAL REPORT CMU/SEI-2003-TR-004 ESC-TR-2003-004, March 2003.

Alternatively you can use an example from a current or recent project.

Tasks

Perform the following tasks. Write down your findings.

1. Define the system context and the actors of the system.

2. Define the user stories for the system

Software Architecture: Unit 1

Exercise 1

Solution

	Web framework	Programming language
Taylor: "principal design decisions"	no	no
Bass: "structure, elements, properties, relationships"	yes	no
Perry, Wolf: "elements, form, relationships"	yes	no
ANSI/IEEE: "organization, components, relationships, principles"	yes	no
Shaw, Garlan 1996: "components, interaction"	yes	no
Kruchten (RUP): "*significant* decisions, organization, elements, interfaces, behavior, collaborations, composition, architectural style"	yes	yes
Fowler: "important stuff"	yes	yes
Ford: "hard to change"	no	yes

Exercise 2

Solution

1. Given an option, Kevlin suggests "to partition and abstract" that issue in question in order reduce the significance of your decision. For example, the architect can propose a framework of strategies for solving a particular algorithmic problem. This keeps that part of the software "soft" meaning that alternatives, when available, can easily be considered.

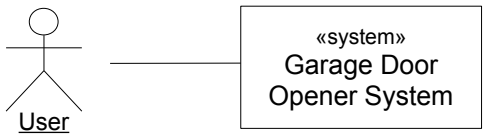
2. Point of uncertainty, and means of deferring it: abstraction, encapsulation, strategies, adapters, layering, virtual machines, ...

Exercise 3

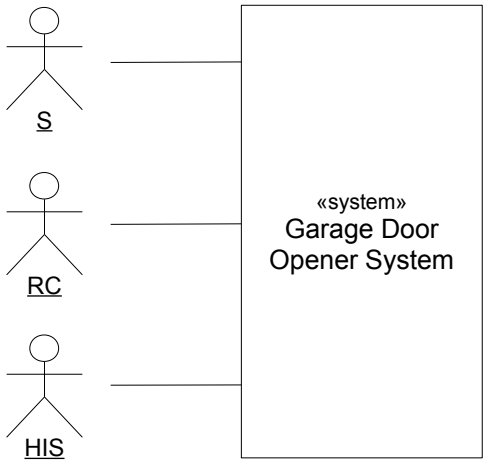
(Sketch of a Solution)

Different Contexts

System contains everything:



Switch (S), Remote Control (RC), and Home Information System (HIS) all are actors:



Use Cases

Use Case 1: Open garage door

Actor	Switch, Remote Control, Home Information System
Main Success Scenario	Actor issues the command “open”. Door opens smoothly from any position until completely opened.
Failure Scenario	While opening the door, an obstacle is detected. Stops the opening of the door within 0.1 seconds.

Use Case 2: Close garage door

Actor	Switch, Remote Control, Home Information System
Main Success Scenario	Actor issues the command “close”. Door closes smoothly from any position until completely closed.
Failure Scenario	While closing the door, an obstacle is detected. Stops the closing of the door within 0.1 seconds.

Use Case 3: Stop opening/closing

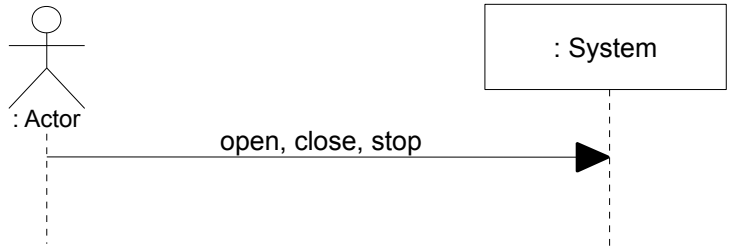
Actor	Switch, Remote Control, Home Information System
Main Success Scenario	Actor issues the command “stop”. When moving, door stops opening or closing (within 0.1 seconds), else nothing.
Failure Scenario	-

Use Case 4: Get diagnostics

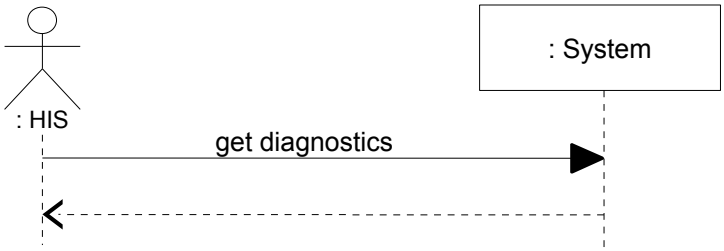
Actor	Home Information System
Main Success Scenario	Actor issues the command “get diagnostics”. System reports on door status. The door's status is on of {OPEN, CLOSED, OPENING, CLOSING, STOPPING, STOPPED_BY_OBSTACLE, STOPPED}.
Failure Scenario	-

System Sequence Diagrams

Trivial for Use Cases 1-3



Still simple for Use Case 4



Software Architecture: Unit 3

Exercise 1

Given the interface of a Stack with invariants, pre- and postconditions:

```
/**
 * Stack with invariants, pre- and post conditions.
 *
 * INV count_is_never_negativ:
 *            getCount() >= 0
 * INV count_is_never_greater_than_capacity:
 *            getCount() <= getCapacity()
 * INV item_at_top:
 *            (getCount() > 0) implies (itemAt(getCount()) == top())
 */
public interface Stack<G> {

        // basic queries
        public int getCount();
        public int getCapacity();

        /**
        * PRE i_small_enough: i > 0;
        * PRE i_big_enough: i <= getCapacity();
        */
        public G itemAt(int i);

        // derived queries
        /**
        *  POST consistent_with_count: @result == (getCount() == 0)
        */
        public boolean isEmpty();

        /**
        *  PRE not_empty: getCount() > 0
        *  POST unchanged_representation: @result == itemAt(count)
        */
        public boolean isFull();

        /**
        *  PRE not_empty: getCount() > 0
        *  POST unchanged_representation: @result == itemAt(count)
        */
        public G top();

        // commands
        // Find pre/post conditions
        public void push(G g);

        // Find pre/post conditions
        public void remove();
}
```

Notice that the implementation of the stack has an arbitrary, but fixed capacity and that itemAt() is 1 based.

Tasks

1. Find appropriate pre- and postconditions for push() and remove().
2. Implement the interface in a class MyStack.

3. Install the cofoja Library from https://github.com/nhatminhle/cofoja/releases and check its usage at https://github.com/nhatminhle/cofoja.
4. Add the invariants, pre- and postconditions using the cofoja Library and Syntax. Where do you add the annotations?
5. Write a client `StackClient` that uses your implementation of the stack. Implement cases that violate pre- and/or postconditions. Execute the client (as follows or in eclipse)

```
java -javaagent:cofoja.asm.jar -ea StackClient
```

(If you have any trouble installing the cofoja library, do at least the first two tasks)

Exercise 2

Consult the following Java interfaces/classes. What principles of good design are followed, which are violated?

- `java.util.Iterator`
- `java.util.Stack`
- `java.lang.String`

Also discuss the dependencies introduced by these classes. Are these dependencies needed?

Take all methods of `java.lang.StringBuilder` and separate them into "query" and "action" methods. Are there any that might fit both?

Exercise 3

Suppose that you write a class that implements the `java.util.Collection` (or similar) interface. The class shall have the set semantics (unique entries only, no ordering). An excerpt for this interface is:

```
public interface Collection<E> {
 public boolean add(E e);
 public boolean remove(E e);
 // other methods not important for this exercise
}
```

Discuss the given methods in the light of the principles of good design.

Now suppose you write a (JUnit) test class which tests your implementation. Are the return values of any help? If the return values were omitted from the interface what additional method or methods would you need to be able to test your implementation?

Software Architecture: Unit 3

Exercise 1

```
// Stack.java
import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;

@Invariant("(getCount() >= 0) && (getCount() <= getCapacity()) && (isEmpty() ||
itemAt(getCount()).equals(top()))")
public interface Stack<G> {

        // basic queries
        @Ensures("result >= 0")
        public int getCount();

        @Ensures("result > 0")
        public int getCapacity();

        @Requires("i > 0 && i <= getCount()")
        public G itemAt(int i);

        // derived queries
        @Ensures("result == (getCount() == 0)")
        public boolean isEmpty();

        @Ensures("result == (getCount() == getCapacity())")
        public boolean isFull();

        @Requires("!isEmpty()")
        @Ensures("result == itemAt(getCount())")
        public G top();

        // commands
        @Requires("g != null && !isFull()")
        @Ensures("(getCount() == old(getCount()) + 1) && top().equals(g)")
        public void push(G g);

        @Requires("!isEmpty()")
        @Ensures("(getCount() == old(getCount()) - 1) && (itemAt(old(getCount())) == null)")
        public void remove();
}

// MyStack.java
import java.lang.reflect.Array;
public class MyStack<G> implements Stack<G> {
    int capacity = 0;
    int currentIndex = 0;
    G elements[];

    public MyStack(Class<G> clazz, int capacity) {
        this.capacity = capacity;
        elements = (G[]) Array.newInstance(clazz, capacity);
    }

    // basic queries
```

```
    @Override
    public int getCount() {
        return currentIndex;
    }

    @Override
    public int getCapacity() {
        return capacity;
    }

    @Override

    public G itemAt(int i) {
        return elements[i-1];
    }

    @Override
    public boolean isEmpty() {
        return currentIndex == 0;
    }

    @Override
    public boolean isFull() {
        return currentIndex == capacity-1;
    }

    @Override
    public G top() {
        return elements[currentIndex-1];
    }

    @Override
    public void push(G g) {
        elements[currentIndex] = g;
        currentIndex++;
    }

    @Override
    public void remove() {
       // set old item to null, so that it can be deleted
       elements[currentIndex-1] = null;
        currentIndex--;
    }
}
```

Exercise 2

`java.util.Iterator`

- Method `E next()` not only returns an element from the collection to iterate, but positions the iterator onto the next element. That is, the method changes the state of the iterator. Hence, it violates the Query Action separation principle.
- Optional method `void remove()` violates the principle of strong cohesion: An iterator should iterate on the elements of a collection, but not modify it or remove elements from it.
- Remark 1: There are certain technical contexts such as transactions which, in turn, favor the availability of a `remove()` method, so the inventors of this interface can defend this choice.
- Remark 2: The API specification of this interface says that the `remove()` method is optional. This is questionable since from a point of view of a client, it cannot know in advance if this method is supported or not.

SRP: no, OCP: ok, LSP: ok, ISP: no, DIP: NA

`java.util.Stack`

- Stack inherits from Vector which in turn implements the List interface. Thus, a Stack instance IS a list, that is, it supports all the List methods.
- Method E pop() violates the Query Action separation principle since not only the top element is returned, but it is also removed from the stack.

SRP: no (search?), OCP: no, LSP: no (Implements List), ISP: no, DIP: NA

`java.lang.String`

- String is an implementation of the Value object pattern. The Value object pattern denotes a class where the attributes of the instances of this class cannot be changed after creation. There are read-only (query) methods only.
- Critics: Some of the (static) methods should (could) be put into one or more separate helper classes.

SRP: no (static methods to helper classes), OCP, LSP, ISP: NA (class is final), DIP: NA

`java.lang.StringBuilder`

- Many methods such as `StringBuilder append(...)` and `StringBuilder insert(...)` return a reference to itself. This is a violation of the Query Action separation principle. However, this allows to program using the "cascading" style:
- `StringBuilder sb = …; sb.append('a').append('b').append('c'); // sb contains "abc"`

Exercise 3

Solution

Given the Query Action separation principle, the given two methods in interface Collection<E> are actions. Hence, the principle is violated.

Returning a Boolean value enables testing without the need of further helper methods, however. For example:

```
@Test
public void testAdd() {
   Collection<Integer> s = new MySetImpl<Integer>();
   assertTrue(s.add(5));
   assertFalse(s.add(5)); // the value 5 is already in  the set
}
```

Similar to method remove().

If the given two methods do not return a Boolean value then testing must be built around methods like boolean contains(E e), boolean isEmpty(), etc.

Software Architecture: Unit 4

Exercise 1: Decorator Pattern

In the example we looked at the decorator pattern employed for the beverages.

Task

- Draw the Class Diagram for the decorator pattern for that example. Include the classes Beverage, Espresso, Decaf, CondimentDecorator, Mocha,WhippedCream
- The coffee has now introduced sizes to their menus, so you can order a coffee in small, medium, and large sizes. The beverage class has been added two methods setSize() and getSize(). The condiments are also charged according to size.
  Alter the decorator classes to handle these requirements.

Exercise 2: Layers

Given two classes, each in a separate layer package. Class A of layer 1 uses class B of layer 2. Hence, layer 1 depends on layer 2:

```
package layer1;
import layer2.B;
public class A {
      private B b;
      public A(B b) { this.b = b; }
      public void service() {
            // use b ...
            String result = b.use();
      }
}
package layer2;
public class B {
      public String use() {
            return "Some string...";
      }
}
```

Task

Introduce an auxiliary layer, say layer 12, which resides in between the two layers 1 and 2, and let layers 1 and 2 depend on the newly introduced layer. What do you put into layer 12? How can layer 1 use an instance of `layer2.B`?

Exercise 3 / Homework

Reading: "Patterns: The Top Ten Misconceptions", by John Vlissides, Object Magazine, March 1997 . See here: http://www.research.ibm.com/designpatterns/pubs/top10misc.html

Task

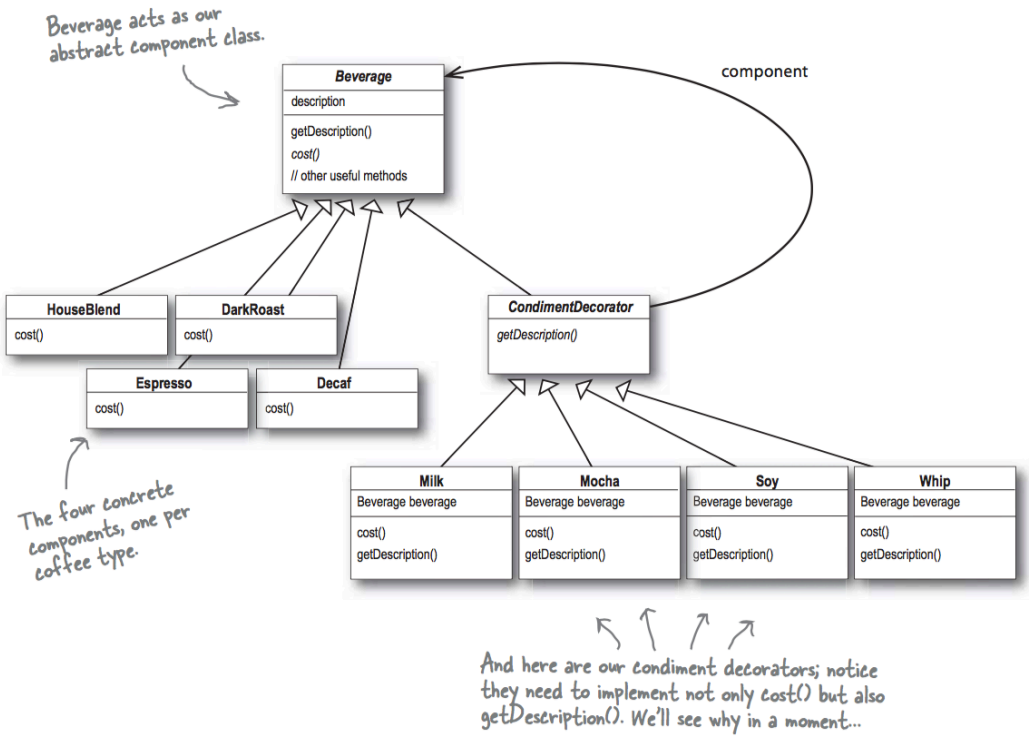
Answer the following questions:

- Given a problem and its solution, why is this not (yet) a pattern?
- According to Vlissides, what are the main benefits of patterns?
- Since patterns don't guarantee anything (Vlissides), what is in your opinion their benefit?
- According to Vlissides, what is the general concept of patterns?

Software Architecture: Unit 3

Exercise 1: Decorator Pattern

1) Class Diagram



2) Sizes

Add getSize() in the Beverage class, implement it as beverage.getSize() in CondimentDecorator. Calculate costs according to size in the overridden cost() methods

Exercise 2: Layers

```
package layer1;
public class A {
      private layer12.IB b;
      public A(layer12.IB b) { this.b = b; }
      public void service() {...}
}

public class Factory { // used in layer1 to provide an instance for IB
      public layer12.IB makeIB() { ... } // uses dynamic features of Java...
}

package layer12;
public interface IB {
      public String useIt();
}
```

```
package layer2;
import layer12.IB;
public class B implements IB {
      public String useIt() {
            return "Some string...";
      }
}
```

Exercise 3 / Homework

Given a problem and its solution, why is this not (yet) a pattern?

- recurrence: the problem must occur in several different situations
- teaching: you must be able to teach the proposed solution
- name: you must give it a name

According to Vlissides, what are the main benefits of patterns?

- Capture of expertise
- Improvement of communication
- Improvement of system understanding and documentation (several classes in one step)

According to Vlissides, what is the general concept of patterns?

- A vehicle to capturing and conveying expertise

Software Architecture: Unit 5

Task

1. Choose the module to decompose.

At the beginning, the only module we have is the System.

2. Refine the module

Modifiability: The control devices used in different products differ. Adaption of the software to a different control device should be done within one person-day.

Concrete Modifiability Scenario :

Part of Scenario	**Possible Values**
Source	developer
Stimulus	wishes to vary the functionality
Artifact	Sytem
Environment	at compile time
Response	makes modification without affecting other functionality
Response measure	effort

Performance Scenario: If an obstacle is detected by the garage door during descent, the door must halt (or re-open) within 0.1 second.

Concrete Performance Scenario :

Part of Scenario	**Possible Values**
Source	from withing the system
Stimulus	sporadic events arrive
Artifact	System
Environment	normal mode
Response	processes stimuli
Response measure	deadline

***Modifiability* tactic sets are** (see slides):

- “localize changes”, “prevent the ripple effects”, “defer binding time”

Since changes will occur during design time → primary tactic sets are: “localize changes”, “prevent the ripple effects”

From these tactic sets we choose:

- “semantic coherence” and “information hiding” and combine them with “generalize modules”

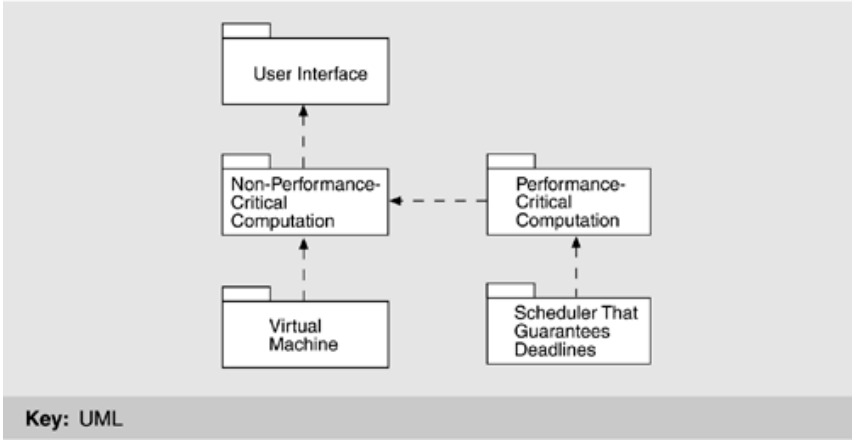
***Performance* tactic sets are**:

- “resource demand”, “resource management”, “resource arbitration”

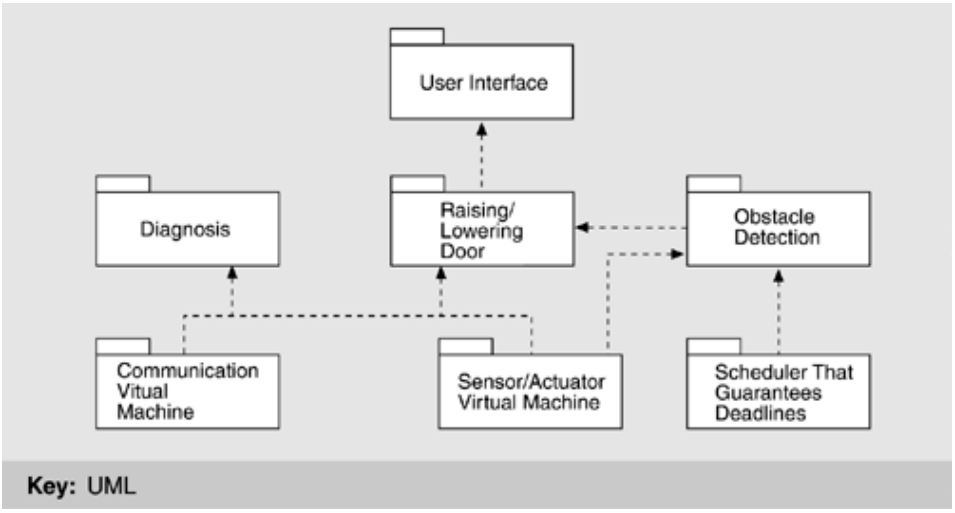
From these, the following *tactics* are chosen:

- “increase computational efficiency”, “introduce concurrency”, and “scheduling policy”

From the tactics above, the following *architectural pattern* can be derived:



The above modularization is coarse-grained, but includes tactics to achieve the garage doors drivers. Next step refines it.



c) In the next step, the modularization is refined. Key modifiability tactics include:

- “abstract common services” → this tactic prescribes new modules comprising common or similar responsibilities that previously resided in multiple modules:
  → virtualization of UI
  → virtualization of sensors/actuators

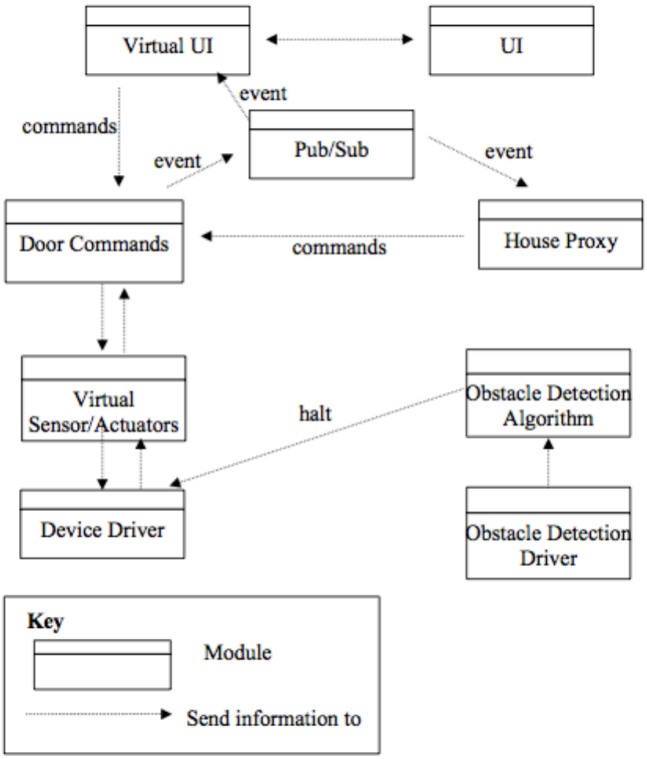
- "break dependency chain"
  → insert an intermediary between publisher and consumer of data
  → Pub/Sub mechanism.

The following modularization is more precisely described in "Identifying aspects using architectural reasoning" by Len Bass et al.[1]

Modules and associated responsibilities:

- UI. Manages interactions with UI sensors and displays. Handles changes in UI devices.
- Virtual UI. Translates interactions from UI int a set of commands common to all UIs. This module hides changes made in UI devices. (This module also checks "security codes" by sending messages to the House proxy.)
- Pup/Sub. Handles events received from modules on a subscription basis. Allows parts of different products to subscribe as needed.
- House proxy. Manages interactions with the HIS if it exists.
- Door commands. Manages all door's actions.
- Virtual sensor/actuator. Hides details of actual sensors or actuators (except for obstacle detection).
- Device driver. Handles interactions with particular sensors or actuators except for obstacle detector sensor.
- Obstacle detection algorithm. Handles obstacle detection. It sends a halt instruction directly to the door motor.
- Obstacle detection driver. Handles interactions with the obstacle detection sensor.

[1] L. Bass, M. Klein, and L. Northrop, "Identifying Aspects using Architectural Reasoning", presented at Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD Conference, 2004.


Virtual UI
UI
event
commands
Pub/Sub
event
event
Door Commands
commands
House Proxy
Virtual
Sensor/Actuators
halt
Obstacle Detection
Algorithm
Device Driver
Obstacle Detection
Driver
Key
Module
Send information to

Software Architecture: Unit 5

Exercise / Homework

The „garage door opener“ example will be used to derive a software architecture using the Attribute Driven Design method. Recall the general requirements:

> The controller for a garage door opener is an embedded real-time system that reacts to “open” and “close” commands from several buttons installed in the house and from a remote control unit, usually located in a car. The controller then controls the speed and direction of the motor, which opens and closes the garage door. The controller also reacts to signals from several sensors attached to the garage door. One of those sensors detects resistance to the door’s movement. If the amount of resistance measured by this sensor is above a certain limit, the controller interprets that resistance as an obstacle between the garage door and the floor. As a reaction, the motor closing the garage door is stopped.
>
> From: „Deriving Architectural Tactics: A Step Toward Methodical Architectural Design“ by Felix Bachmann, Len Bass, and Mark Klein; TECHNICAL REPORT CMU/SEI-2003-TR-004 ESC-TR-2003-004, March 2003.

Quality Requirements

Beside the functional requirements (see above and also Exercises of Unit 1), the following quality requirements in terms of quality scenarios exist:

1. **The control devices for opening and closing the door are different for various products of the product line. These include controls such as various kinds of remote controls, home information systems, and switches located near the doors.**
2. The processor used in different products will differ.
3. **If an obstacle is detected by the garage door during descent, it must halt (alternatively re-open) within 0.1 second.**
4. The garage door opener must be accessible for diagnosis and administration from within a home information system using a product-specific diagnosis protocol.

The quality scenarios **in bold** will be subject of the task below.

Task

Given the ADD steps, derive a software architecture which respects the above given quality requirements. (For the sake of this exercise, look at quality scenarios no. 1 and 3 only.) These steps are:

1. Choose the module to decompose.
2. Refine the module according to these steps:
    a. Choose the architectural drivers from the quality scenarios and functional requirements.
    Hint:
    For quality scenario 1, generate the Concrete Scenario (see corresponding tables below).
    For quality scenario 3, generate the Concrete Scenario (see corresponding tables below).
    b. Choose an architectural pattern that satisfies the architectural drivers. Select the pattern based on a tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.
    c. Instantiate modules and allocate functionality.
    d. Define interfaces for the child modules. The decomposition provides modules and constraints on the types of module interactions.
    e. Verify and refine the use cases and quality scenarios, make them to be constraints for the child modules.

3. Repeat the steps above for every module that needs further decomposition.

Auxiliary Tables

Modifiability

Modifiability General Scenario:

Part of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artifact	System, user interface, platform, environment; system that interoperates with target system
Environment	At runtime, compile time, build time, design time
Response	Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
Response measure	Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes

Concrete Modifiability Scenario (to be completed):

Part of Scenario	Possible Values
Source	
Stimulus	
Artifact	
Environment	
Response	
Response measure	

Performance

Performance General Scenario:

Part of Scenario	Possible Values
Source	One of a number of independent sources; possibly from within system
Stimulus	Periodic events arrive; sporadic events arrive; stochastic events arrive
Artifact	System
Environment	Normal mode; overload mode
Response	Processes stimuli; changes level of service
Response measure	Latency, deadline, throughput, jitter, miss rate, data loss

Concrete Performance Scenario (to be completed):

Part of Scenario	Possible Values
Source	
Stimulus	
Artifact	
Environment	
Response	
Response measure	