

CSC 7322 : Object Oriented Development

J Paul Gibson, A207

`paul.gibson@int-edu.eu`

<http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/>

After OO?

[.../~gibson/Teaching/CSC7322/L7-AfterOO.pdf](http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/L7-AfterOO.pdf)

Some future (current) trends

Various improvements to *make OOP better*:

- Aspects
- Static Analysis – ESC Java
- Emphasis formality/modelling – Java JML
- Features
- Services
- Components
- Parallel Programming
- Self -*?

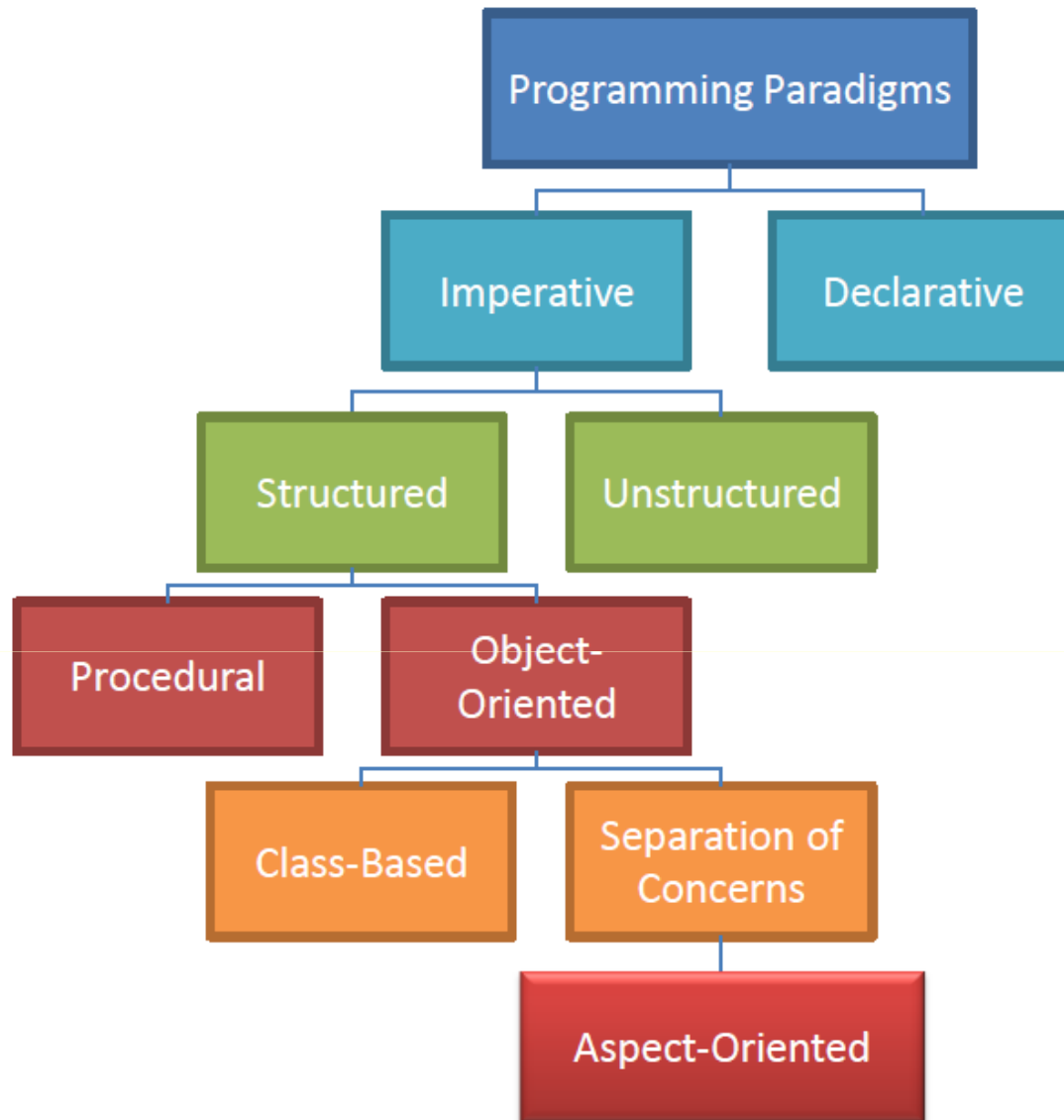
Aspects – Overview of problem

Some aspects of code can't be modularized

- They cross-cut the modularization boundaries
- They appear everywhere in the code
- Even if you use good Object-Oriented Design
- Even if you use a well-known pattern like MVC
- Development of these aspects is hard
- Maintenance of these aspects is also hard
- An example is logging

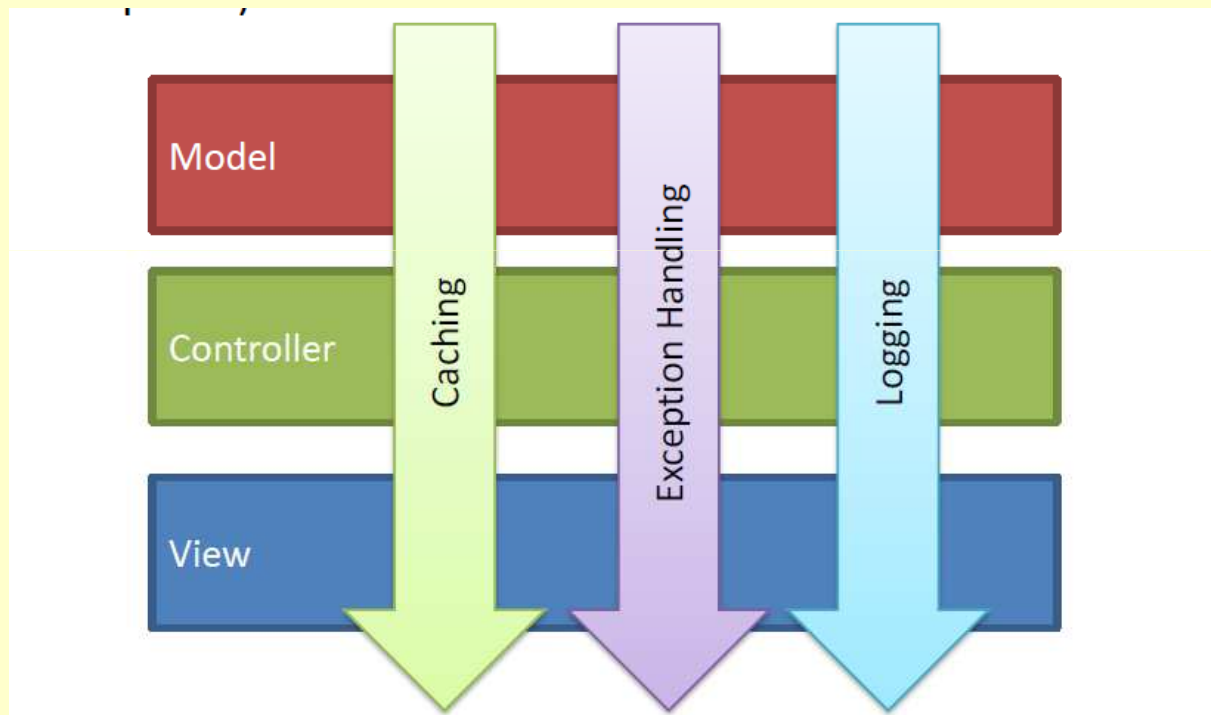
Aspect-Oriented Programming (AOP)

- Yet another programming approach
- Isolates cross-cutting/supporting functions
- Improves the modularity, reusability, and separation of concerns
- Doesn't replace existing programming techniques
- Works with existing programming approaches
- Breaks the program into concerns
- The basis of Aspect-Oriented Software Development (AOSD)
- Created by Gregor Kiczales at Xerox PARC
- Mainly promoted by the Java community
- AspectJ as the flag of AOP frameworks



Cross-Cutting Concern

- Cannot be separated into a module
- Can be a challenge to separation of concerns
- Can easily be detected with static analysis tools (i.e. NDepend)



Some Cross-Cutting Concerns

- Logging
- Thread Synchronization
- Caching
- Lazy Loading
- Transaction Management
- Data Binding
- Exception Handling
- Validation
- Dependency Injection
- Security
- Authentication and Authorization
- Property Changes

Aspect

- A special kind of module
- Handles cross-cutting concerns
- Encapsulates cross-cutting concerns
- Incorporates them into the system
- Can be implemented based on the join-point model

Potential Issues with AOP

- The understanding of cross-cutting concerns
- The understanding of the flow of code
- Simple mistakes as big problems
- Debugging AOP code
- Efficiency

Principles of AOP

- **Join-point**

The points/events in codebase *when a code is executing*.

A method or constructor is called.

A method or constructor is executed.

A setter or getter method execution.

An exception handler is executed.

- **Point-cut**

A set of join-points using set theory operations.

- **Advice**

Additional behavior attached to a point-cut.

Misconceptions

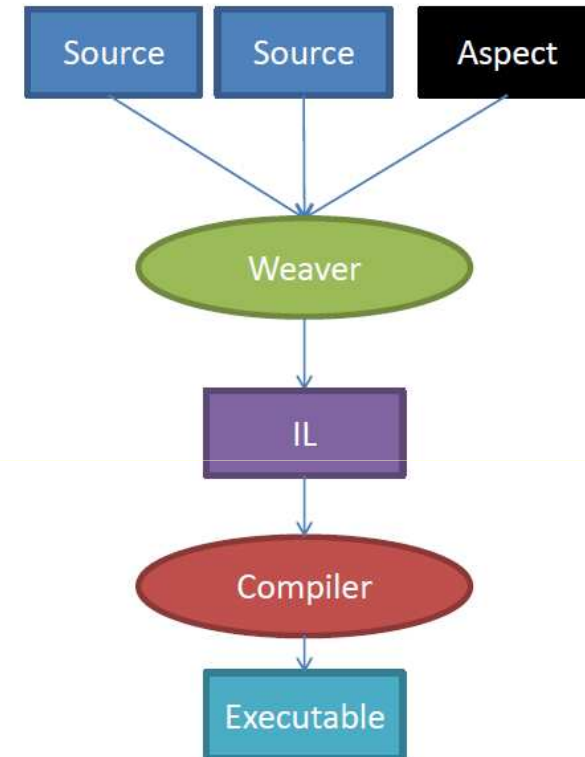
- AOP is not a replacement for current techniques.
- AOP is not a separate approach of programming.
- Aspects are not infrastructure or auxiliary parts of code.
- AOP is not only for applying changes to existing code.
- Good use of OOP doesn't eliminate the need for AOP.
- AOP code is not hard to read.
- AOP is not for logging only.

Projects Using AOP

- Spring Framework
- JBoss Application Server
- Glassbox Troubleshooting Agent
- Tapestry
- Contract4J
- Jakarta Cactus
- Objectweb Cooperative Open Workflow
- Objectweb Java Open Transaction Manager
- OpenLogic
- MySQL Connector/J

Language Support for AOP

- AOP support as a built-in feature of the language
- Based on Method Call Interception
- Providing a stub/proxy to call for each method
- Weaving transforms the code
- Weaver gets Object-Oriented code and integrates aspects into it
- Two weaving approaches:
 1. Runtime/Dynamic Weaving
 2. Compile-time/Static Weaving



AOP Frameworks

Java

- AspectJ
- JAC
- dynaop
- AspectWekz

.NET

- LinFu
- PostSharp
- Spring.NET
- LOOM.NET
- NAspect

Aspect J

AspectJ is an aspect-oriented extension created at PARC for the Java programming language.

It is available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse.

AspectJ has become the widely-used de-facto standard for AOP by emphasizing simplicity and usability for end users.

It uses Java-like syntax and has included IDE integrations for displaying crosscutting structure since its initial public release in 2001.

Aspects are similar to classes because...

- aspects have type
- aspects can extend classes and other aspects
- aspects can be abstract or concrete
- non-abstract aspects can be instantiated
- aspects can have static and non-static state and behavior
- aspects can have fields, methods, and types as members
- the members of non-privileged aspects follow the same accessibility rules as those of classes

Aspects are different than classes because...

- aspects can additionally include as members pointcuts, advice, and inter-type declarations;
- aspects can be qualified by specifying the context in which the non-static state is available
- aspects can't be used interchangeably with classes
- aspects don't have constructors or finalizers, and they cannot be created with the new operator; they are automatically available as needed.
- privileged aspects can access private members of other types

Static Analysis – ESC Java

Extended static checking: a range of techniques for statically checking the correctness of various program constraints... usually involves the use of an automated theorem prover

The "Extended Static Checker for Java," is a programming tool that attempts to find common run-time errors in Java programs at compile time. This technique was pioneered in ESC/Java and can be thought of as an extended form of type checking.

For ESC/Java, the Simplify theorem prover was used.

ESC/Java is neither sound nor complete. This was intentional.

Question: can you think why?

ESC/Java was originally developed at the Compaq Systems Research Center (SRC).

Recent versions of ESC/Java are based around the Java Modeling Language (JML).

Static Analysis – ESC Java

```
class Bag {
    int[] a;
    int n;

    Bag(int[] input) {
        n = input.length;
        a = new int[n];
        System.arraycopy(input, 0, a, 0, n);
    }

    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 1; i <= n; i++) {
            if (a[i] < m) {
                mindex = i;
                m = a[i];
            }
        }
        n--;
        a[mindex] = a[n];
        return m;
    }
}
```

QUESTION: In the Bag class what potential problems could an ESC provide warnings for?

Simplest Usage: To check our definition of class Bag with ESC/Java, we type the following similar command line:

```
escjava Bag.java :
```

```
Bag.java:6: Warning: Possible null dereference (Null)
```

```
    n = input.length;  
      ^
```

```
Bag.java:15: Warning: Possible null dereference (Null)
```

```
    if (a[i] < m) {  
        ^
```

```
Execution trace information:
```

```
    Completed 0 loop iterations in "Bag.java", line 14, col 4. Bag.java:15: Warning:
```

```
Array index possibly too large (IndexTooBig)
```

```
    if (a[i] < m) {  
        ^
```

```
Execution trace information:
```

```
    Completed 0 loop iterations in "Bag.java", line 14, col 4.
```

```
Bag.java:21: Warning: Possible null dereference (Null)
```

```
    a[mindex] = a[n];  
          ^
```

```
Execution trace information:
```

```
    Completed 0 loop iterations in "Bag.java", line 14, col 4.
```

```
Bag.java:21: Warning: Possible negative array index (IndexNegative)
```

```
    a[mindex] = a[n];  
          ^
```

```
Execution trace information:
```

```
    Completed 0 loop iterations in "Bag.java", line 14, col 4.
```

```
5 warnings
```

Remarks

Actually, the previous output was generated by the command

```
escjava -quiet Bag.java
```

The normal output of ESC/Java also includes various progress messages and timing information which are omitted here.

Some of the messages above include a part marked as ``Execution trace information".

NOTE: We now must decide what to do about this warning.

QUESTION: what are the non-ESC options?

What to do about the warning: ESC/Java's first warning is that the attempt to access `input.length` might fail because `input` might be null.

One approach would be to decide that the implementation of the constructor is incorrect. Following this approach, we would modify the constructor to test for a null argument and, for example, construct an empty multiset:

```
Bag(int[] input) {
    if (input == null) {
        n = 0;
        a = new int[0];
    } else {
        n = input.length;
        a = new int[n];
        System.arraycopy(input, 0, a, 0, n);
    }
}
```

This would indeed eliminate the first warning. ... **but?**

More advanced usage: We write a **requires (precondition) *pragma*** for the **Bag constructor**.

We decide that the implementation of the constructor is correct, but that we do not intend for the constructor ever to be called with a null argument.

We inform ESC/Java of this decision by adding an annotation to the constructor declaration:

```
//@ requires input != null;
    Bag(int[] input) {
        n = input.length;
        ...
    }
```

When the character @ is the first character after the initial // or /* of a Java comment, as in the first line of the program fragment above, ESC/Java expects the body of the comment to consist of a sequence of ESC/Java annotations, known as *pragmas*. The *requires pragma* above specifies a *precondition* for the constructor, that is, a boolean expression which must be true at the start of any call.

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

When checking code that calls a method or constructor annotated with a precondition, ESC/Java will issue a warning if it cannot confirm that the precondition (with the values of the actual parameters substituted for the formal parameter names) would always evaluate to true at the call site.

QUESTION: What can we do about the second warning:

```
Bag.java:15: Warning: Possible null dereference (Null)
    if (a[i] < m) {
```

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

QUESTION: What can we do about the second warning:

```
Bag.java:15: Warning: Possible null dereference (Null)
    if (a[i] < m) {
```

We could deal with this warning by either of the approaches discussed above in connection with the first warning--that is,

by adding:

```
the precondition requires a != null to the extractMin method,
```

```
(or by adding special code for the case a == null.)
```

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

However ESC/Java offers yet another choice, which is to specify that the a field of any Bag is always supposed to be non-null.

To do this, we annotate the declaration of a with a non_null pragma:

```
class Bag {  
    /*@ non_null */ int[] a;  
    ...  
}
```

This causes ESC/Java to assume that the a field of any Bag object is itself non-null (and thus can safely be dereferenced). Conversely, it causes ESC/Java to issue a warning for any assignment to the a field of a Bag, if it cannot confirm that the expression being assigned will have a non-null value at run time. Furthermore ESC/Java will check that every Bag constructor initializes the a field of the constructed object to a non-null value.

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

QUESTION: What can we do about the third warning:

```
Bag.java:15: Warning: Array index possibly too large
(IndexTooBig)
    if (a[i] < m) {
        ^
```

Examining the program, we now find a genuine bug.

The for loop starting on line 15 (in the original program) examines array elements a[1] through a[n], but array indexing in Java is zero based.

We correct the line to read

```
for (int i = 0; i < n; i++) {
```

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

QUESTION: What can we do about the fourth warning:

The fourth warning

```
Bag.java:21: Warning: Possible null dereference (Null)
    a[mindex] = a[n];
                ^
```

Now requires no action, as the `non_null` pragma we added earlier already prevents `a` from being null. In other words, the second and fourth warnings complain about the same problem.

Remark: ESC/Java often avoids issuing such redundant warnings. Note, for example, that it doesn't complain about the expression `a[m]` on (original) line 17, or the expression `a[mindex]` on the left hand side of the assignment on (original) line 21. However, it does not avoid them in all cases.

More advanced usage: **We write a requires (precondition) *pragma* for the Bag constructor.**

QUESTION: What can we do about the fifth warning:

```
Bag.java:21: Warning: Possible negative array index  
(IndexNegative)
```

```
    a[mindex] = a[n];  
                ^
```

The problem is that the code in (original) lines 20-21 removes an element from the bag even when the bag is already empty (that is, when `this.n == 0` on entry to `extractMin`).

ESC/Java has called our attention to the need for another design decision: do we add special code to handle the situation when `extractMin` is called on an empty bag, or do we add a precondition forbidding such calls?

Let's try the latter course:

```
    //@ requires n >= 1;  
    int extractMin() {
```

Static Analysis – ESC Java

With all the changes described above, our example program is now:

```
class Bag {
  /*@ non_null */ int[] a;
  int n;

  //@ requires input != null;
  Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
  }

  //@ requires n >= 1;
  int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for (int i = 0; i < n; i++) {
      if (a[i] < m) {
        mindex = i;
        m = a[i];
      }
    }
    n--;
    a[mindex] = a[n];
    return m;
  }
}
```

We now run ESC/Java again and it produces the following warning:

Bag.java:17: Warning: Array index possibly too large (IndexTooBig)
if (a[i] < m) {
 ^

QUESTION: Can you think about why this happens and what we should do about it?

SOLUTION: We supply an **invariant** pragma relating n to $a.length$.

It may appear that ESC/Java is still complaining about the bug we thought we'd fixed earlier, but further study reveals a different problem: ESC/Java has no reason to expect that n , which we intend to be the length of the meaningful prefix of a , will in fact be at most $a.length$.

We can express this intention with an invariant pragma:

```
class Bag {
  /*@ non_null */ int[] a;
  int n;
  //@ invariant 0 <= n & n <= a.length;
  ...
}
```

Roughly speaking, ESC/Java treats an invariant as an implicit postcondition of every constructor and as both a precondition and postcondition of every method.

The semantics of invariant pragmas--and all other ESC/Java pragmas-- can be seen in most standard texts

Java JML -

Basic Approach “Eiffel + Larch for Java”

- Hoare-style (Contracts).
- Method pre- and postconditions.
- Invariants.

Handles:

Inter-module interfaces.
Classes and interfaces.
Data (fields)
Methods.

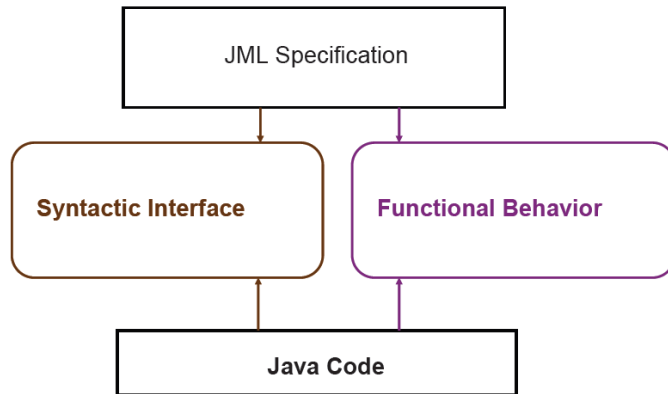
Doesn't handle:

User interface.
Architecture, packages.
Dataflow.
Design patterns

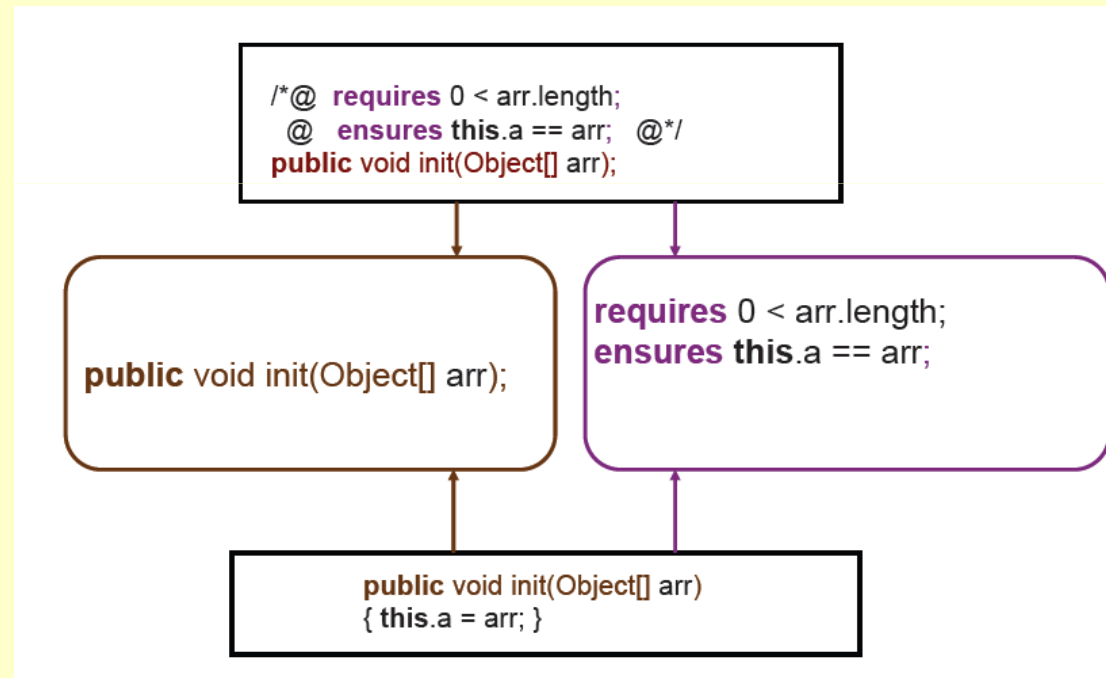
A simple example:

```
public class ArrayOps {  
  
    private /*@ spec_public @*/ Object[] a;  
  
    //@ public invariant 0 < a.length;  
  
    /*@ requires 0 < arr.length;  
    @ ensures this.a == arr;  
    @*/  
    public void init(Object[] arr) {  
        this.a = arr;  
    }  
}
```

Interface Specification



Linking Design and Spec



Like But ... For Java and ...

VDM, but

- OO features

Eiffel, but

- Features for formal verification

Spec#, but

- Different invariant methodology
- More features for formal verification

Unlike OCL and Z

More Java-like syntax.

Tailored to Java semantics.

Many Tools, One Language

Field Detail

SATURATED

public static final int SATURATED;

Method Detail

adjusted

public void adjusted(int amount)

Spec2Class

requires 0 <= this.amount() &&& reduced < 120;
 ensures this <= 100;
 returns this <= 100 &&& amount < 1000;

getRed

public int getRed()

Specifications: post
amount <= 100 &&& this <= 100

Package [Classes](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

JML Annotated Java

```
public class ArrayOps {
    private /*@ spec_public @*/ Object[] a;
    //@ public invariant 0 < a.length;

    /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
    public void init(Object[] arr) {
        this.a = arr;
    }
}
```

javadoc

Web pages

jmlunit

Unit tests

jmlc

Class file

XVP

ESC/Java2

Warnings

Daikon

Data trace file

Bogor

Model checking

JACK, Jive, Krakatoa,
KeY, LOOP

Correctness proof

How Tools Complement Each Other

Different strengths:

1. Runtime checking — real errors.
2. Static checking — better coverage.
3. Verification — guarantees.

Usual ordering:

1. Runtime checker (jmlc and jmlunit).
2. Extended Static Checking (ESC/Java2).
3. Verification tool (e.g., KeY, JACK, Jive).

A classic example: Bounded Stack

```
public class BoundedStack {  
  
    private /*@ spec_public nullable @*/  
        Object[] elems;  
    private /*@ spec_public @*/ int size = 0;  
  
    //@ public invariant 0 <= size;  
    /*@ public invariant elems != null  
        @    && (\forall int i;  
        @          size <= i && i < elems.length;  
        @          elems[i] == null);  
    @*/  
  
    /*@ requires 0 < n;  
        @ assignable elems;  
        @ ensures elems.length == n;  
    @*/  
    public BoundedStack(int n) {  
        elems = new Object[n];  
    }  
}
```

BoundedStack's push Method

```
/*@ requires size < elems.length-1;
   @ assignable elems[size], size;
   @ ensures size == \old(size+1);
   @ ensures elems[size-1] == x;
   @ ensures_redundantly
   @     (\forall int i; 0 <= i && i < size-1;
   @         elems[i] == \old(elems[i]));
   @*/
public void push(Object x) {
    elems[size] = x;
    size++;
}
```

BoundedStack's pop Method

```
/*@ requires 0 < size;
   @ assignable size, elems[size-1];
   @ ensures size == \old(size-1);
   @ ensures_redundantly
   @     elems[size] == null
   @     && (\forall int i; 0 <= i && i < size-1;
   @         elems[i] == \old(elems[i]));
   @*/
public void pop() {
    size--;
    elems[size] = null;
}
```

BoundedStack's top Method

```
/*@ requires 0 < size;  
   @ assignable \nothing;  
   @ ensures \result == elems[size-1];  
   @*/  
public /*@ pure @*/ Object top() {  
    return elems[size-1];  
}  
}
```

Features

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems

A **feature** is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option.

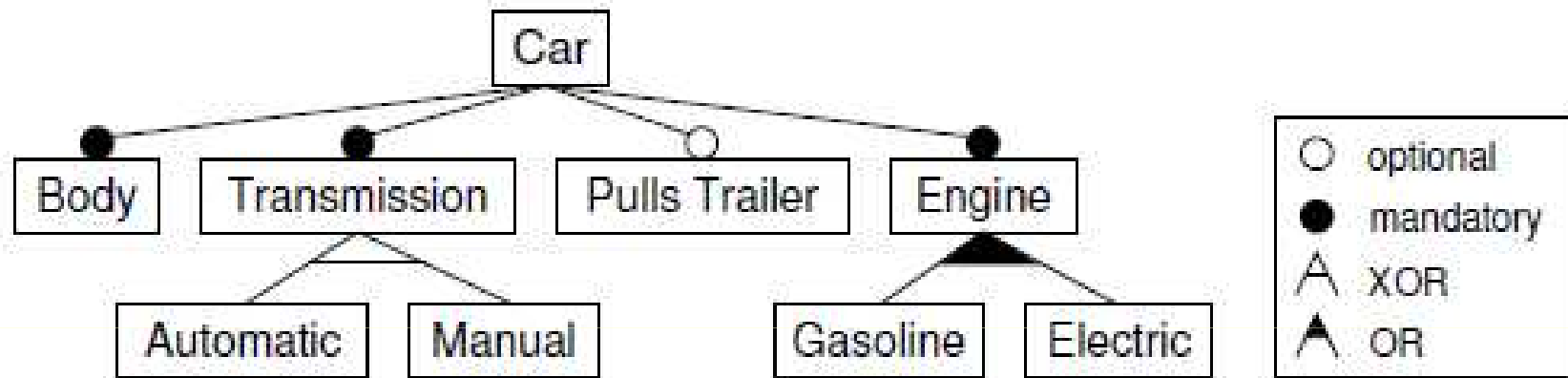
The basic idea of FOSD is to decompose a software system in terms of the features it provides. The goal of the decomposition is to construct well-structured software that can be tailored to the needs of the user and the application scenario.

Typically, from a set of features, many different software systems can be generated that share common features and differ in other features.

The set of software systems generated from a set of features is also called a **software product line**

Features

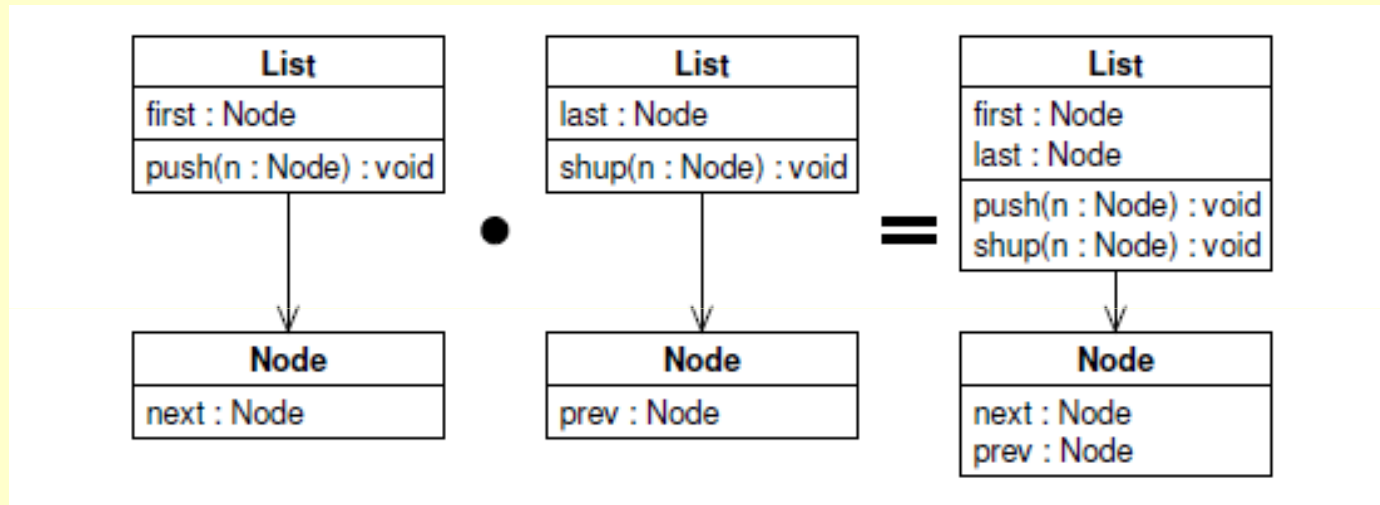
A car example:



Fundamental issue: *feature interactions*

Features

A linked list example:



Feature composition involves merging models

Services

SOA solutions are composed of reusable services based on standards-based, well-defined interfaces.

Web Services currently provide the main enabling technique for Service Oriented Architecture. The Web Service technique can function both as a middleware and a modeling and management tool for composed business processes.

How do you analyze SOA architecture - break down the architecture to its component parts, working from the most primitive to the most sophisticated, testing each component, then the integration of the holistic architecture

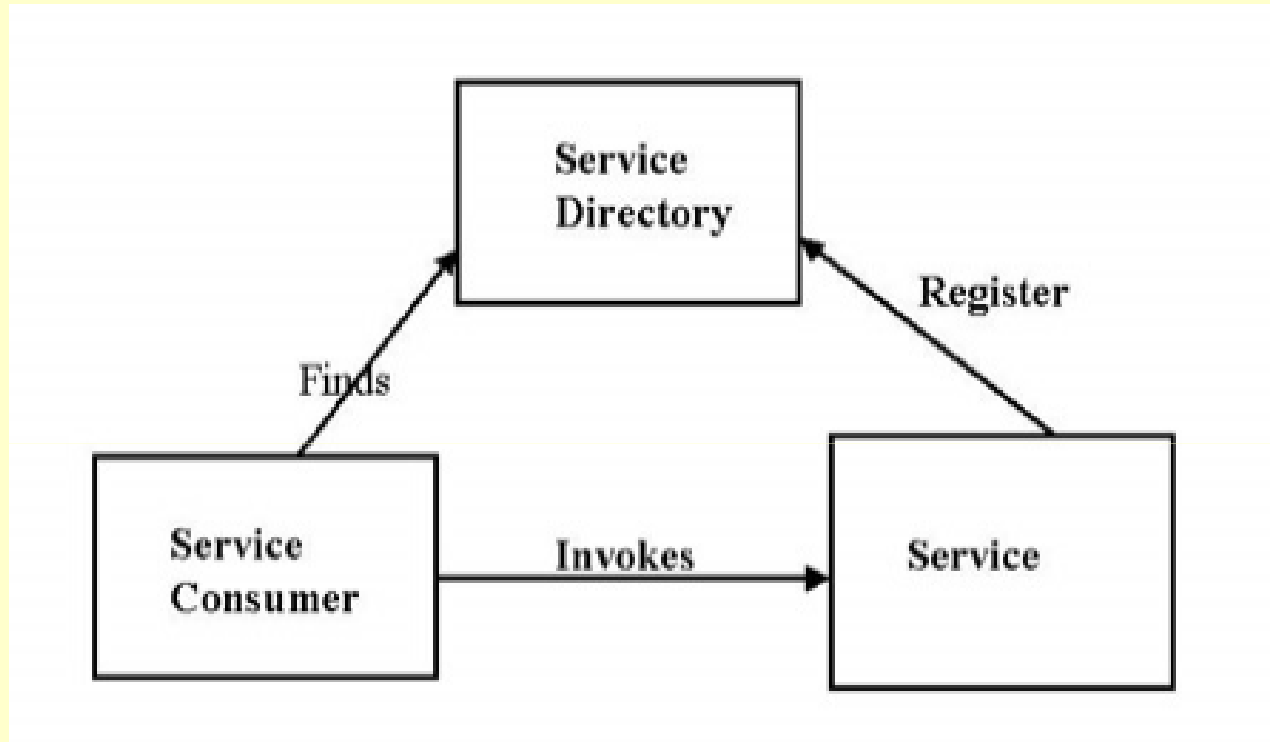
You have to divide the architecture into domains, such as services, security, and governance and test each domain separately using the recommended approach and tools.

Is SOA really new?

The concept of a service is nothing new, but the notion of an SOA has evolved over the past couple of years. It's an architectural style of building software applications that promotes loose coupling between components so that you can reuse them. Thus, it's a new way of building applications with the following characteristics:

- Services are software components that have published contracts/interfaces; these contracts are platform-, language-, and operating-system-independent. XML and the Simple Object Access Protocol (SOAP) are the enabling technologies for SOA, since they're platform-independent standards.
- Consumers can dynamically discover services.
- Services are interoperable.

What is SOA?



QUESTION: What has this got to do with the cloud?

Components

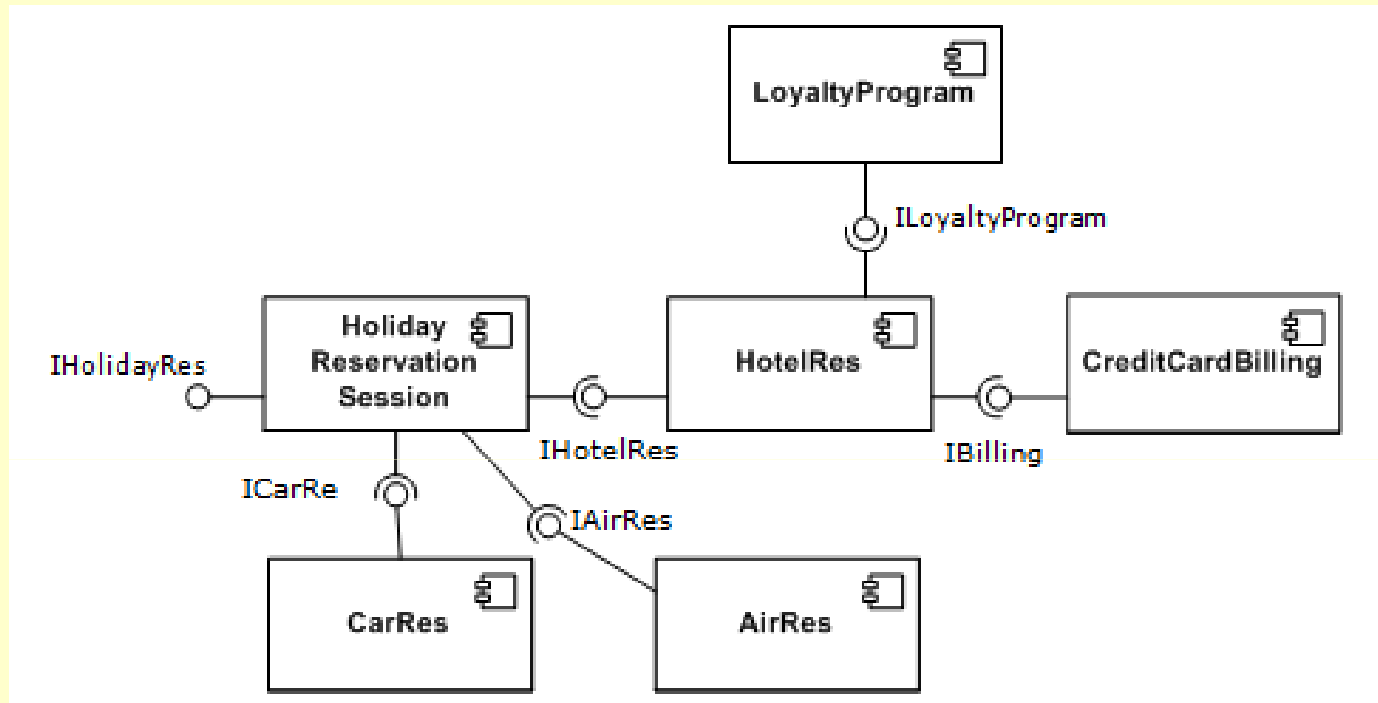
Component-based software engineering (CBSE) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system.

Software engineers regard components as part of the starting platform for service-orientation.

Components play this role, for example, in Web Services, and more recently, in Service-Oriented Architecture (SOA) - whereby a component is converted into a service and subsequently inherits further characteristics beyond that of an ordinary component.

Components can produce events or consume events and can be used for event driven architecture (EDA).

Components: example



Parallel Programming, eg JAVA:

Parallel Java (PJ) is an API and middleware for parallel programming in 100% Java on shared memory multiprocessor (SMP) parallel computers, cluster parallel computers, and hybrid SMP cluster parallel computers.

QUESTION: What is new about the multicore challenge?

Self-*

Self healing

Self managing

Self stabilizing

Self modifying

Self organizing

QUESTION: Does this require self-aware systems ??