Compositional Programming

Oscar Nierstrasz

Software Composition Group
scg.unibe.ch

http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=12511
Roadmap

Early history
Objects
Components
Features
Metaprogramming
Conclusions
Roadmap

Early history

Objects

Components

Features

Metaprogramming

Conclusions
David Wheeler is credited with the invention of the “closed subroutine”. Dijkstra points to this as one of the most fundamental contributions to PL design.

Wheeler is often quoted as saying "Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem."

Another quotation attributed to him is "Compatibility means deliberately repeating other people's mistakes."

Libraries — FORTRAN II (1958)

Large reusable libraries of scientific functions led to the long-term success of FORTRAN

User subroutines are introduced in FORTRAN II (1958).
Recursion — ALGOL (1958)

ALGOL brought recursion into the mainstream

FORTRAN did not support reentrant procedures. ALGOL introduced a run-time stack to support recursive procedures, but the impact was only realized later.
Modules — COBOL (1959)

Modules enabled the stepwise decomposition of large software systems.

Cobol tried to be readable (for managers) but ended up just being verbose. Still the most widely used PL today. Main innovation was in supporting modular programming.
Summary

**Paradigm:** procedural composition

**Motivation:** code reuse, managing complexity
Roadmap

Early history

**Objects**

Components

Features

Metaprogramming

Conclusions
Data Abstraction

**Abstraction** = *elimination of inessential detail*

**Encapsulation** = *bundling operations to access related data as a data abstraction*

**Information hiding** = *providing only the information a client needs to know*

In object-oriented languages we can implement data abstractions as classes.


These three concepts are often confused, but in fact any one may be present without the other two.

See also: William Cook, OOPSLA 2009 for a discussion on the distinction between data abstractions and abstract data types.
Object-Oriented Programming (1962)

OOP was introduced in Simula as an extension to ALGOL to model simulations.

OOP introduces objects, class and inheritance. Objects are composed of subobjects, and subclasses specializations of superclasses.
Smalltalk (1972)

In “pure” OOP, objects are used to model all aspects of design.

Everything is an object

Everything happens by sending messages

5 factorial → 120

```
Integer » factorial

self = 0 ifTrue: [ ^ 1 ].
self > 0 ifTrue: [ ^ self * (self - 1) factorial ].
self error: 'Not valid for negative integers'
```

Smalltalk was the first language to use objects as the only basis for programming. It was inspired by the need for a new language and run–time system needed for the next generation of interactive workstations.

Software entities should be open for extension, but closed for modification.

“In other words, we want to be able to change what the modules do, without changing the source code of the modules.”


Example: Class — instantiate as an encapsulated object; extend as a subclass. Component — fixed interface; hooks to plug in new behaviour (cf eclipse)
Design by Contract

Services should specify clear contracts

“If you promise to call S with the precondition satisfied, then I, in return, promise to deliver a final state in which the post-condition is satisfied.”

If the precondition fails, it is the client’s fault.

If the postcondition fails, it is the supplier’s fault.
If the class invariant fails, it is the supplier’s fault.

See also: [http://www.objectmentor.com/resources/articles/ocp.pdf](http://www.objectmentor.com/resources/articles/ocp.pdf)

DbC is one of the foundations of OO design. It simplifies design decisions, and leads to the development of more robust software by formalizing the expectations of clients and suppliers of services.
Principle of Substitutability

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

Wegner and Zdonik made a first attempt to formulate the notion of “plug compatibility” between objects. The notion is still rather informal — Is a Circle an Ellipse? Depends on the contract clients expect! (Ditto for Square and Rectangle.)
Liskov substitution principle

Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be true for objects $y$ of type $S$, where $S$ is a subtype of $T$.

Restated in terms of *contracts*, a derived class is substitutable for its base class if:

- *Its preconditions are no stronger than the base class method.*
- *Its postconditions are no weaker than the base class method.*


http://www.cse.ohio-state.edu/~neelam/courses/788/lwb.pdf

Note that Liskov and Wing actually refer to a much stronger notion of behavioral substitutability than Uncle Bob (or Wegner and Zdonik do), and is much stronger than what OO programs usually require. It all depends on how strong your type system is!
Polymorphism is often confused with “dynamic binding”. It simply means that entities may have many types, whereas in monomorphic languages (like Pascal or C) entities have unique types.

Universal polymorphism means one function accepts many types. With ad hoc polymorphism, there are actually many functions with the same name.

Java and C++ support all four kinds of polymorphism.

Polymorphism is useful because it enables generic client code to be written that does not depend on the concrete type of the service provided.
A client who expects the behaviour of $f_X$, and applies $f_Y$ to a value in $D_X$ might get a run-time type error.


Covariance intuitively makes sense but is unsafe. It is supported in Eiffel, but is caught at runtime (“CAT-calls” — covariant argument type).
A contravariant result type guarantees that the client will receive no unexpected results.

But contravariance runs counter to intuition, so is rarely used in real languages. Java requires that overridden methods have the same signature. Methods with the same name but different signatures are actually overloaded, opening a whole other can of worms.

Classes are *generators* of types.

“Asteroids are nested volumes in the space of types. Types are points at the apex of each bounded volume.”

Classes are generators of types, and represent families of types.

A specific object has a given, specific type. A class may instantiate a given object, but may also be used to generate subclasses, so it represents a whole family of types.
Frameworks reverse the usual flow of control: you don’t call them; they call you!
The problem with white box frameworks is that the contracts for subclassing the Framework classes are implicit
Design Patterns

Patterns document common language-independent solutions to design problems.

Most of the GOF patterns achieve flexibility by adding a level of indirection.


Most patterns define a set of collaborating roles, which are to be played by objects of your concrete design.
Summary

Paradigm: object composition + incremental refinement

Motivation: domain modeling
Roadmap

Early history

Objects

Components

Features

Metaprogramming

Conclusions
A SPL is a domain specific framework for producing a range of related applications. Key concern: managing variation. Can use a range of techniques.
A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

A software component can be deployed independently and is subject to composition by third parties.


NB: This definition emphasizes composition, not refinement.
But what is a software component?

an object
a class
a template
a method
a procedure
a mixin
a trait
a module
a package
a subsystem
a framework
a script
a service
a metaobject
a metaclass
a design pattern

...?

it depends ...
Applications = Components + Scripts

Components both *import* and *export* services

Scripts *plug* components together

A **scripting language** is a dedicated language for orchestrating a set of tasks (or components).


This definition emphasizes the need to configure components. Components should be plug-compatible, so they are plugged — not wired — together. Sometimes “glue” is additionally needed to adapt components to fit together.
A DSL is a dedicated programming or specification language for a specific problem domain.

SQL is dedicated to the domain of manipulating tables. The Bourne shell is dedicated to scripting Unix commands. Neither language can be used without the components it is intended to script.
Mondrian is a tool for scripting visualizations of software models. In fact it is a component framework, and scripts are simply Smalltalk code using the framework API, but have the flavor of a DSL. This is an “internal” or “embedded” DSL.
The API is designed as a “fluent interface” so code that uses it resembles a DSL. The second example shows how this can be done in Java.

```
mail()
  .from("build@example.com")
  .to("example@example.com")
  .subject("build notification")
  .message("some details about build status")
  .send();
```
Piccola is a minimal language for defining plugs, connectors and scripts.

Piccola was designed as a “pure composition language” for defining “styles” (fluent interfaces), and scripts using those styles. The core paradigm is of communicating agents. Forms are first-class environments to control the scope of scripts.
Piccola

\[ A, B, C \ ::= \ \epsilon \quad \text{empty form} \]
\[ x \rightarrow \quad \text{bind} \]
\[ x \quad \text{variable} \]
\[ A; B \quad \text{sandbox} \]
\[ \text{hide}_x \quad \text{hide} \]
\[ A \cdot B \quad \text{extension} \]
\[ \lambda x.A \quad \text{abstraction} \]
\[ A B \quad \text{application} \]
\[ R \quad \text{current root} \]
\[ L \quad \text{inspect} \]
\[ A | B \quad \text{parallel} \]
\[ \nu c.A \quad \text{restriction} \]
\[ c? \quad \text{input} \]
\[ c \quad \text{output} \]

Piccola is an extension of the pi calculus with first class environments.
Piccola is implemented in Java, and can be used to script Java components (objects) adhering to a particular fluent interface. In the “hello world” example, a button widget is bound to an action that prints “hello world”.

For scripting components written in Java.
Plug-in Architectures

Plug-ins allow you to extend and (sometimes) configure the host application.

Best known examples: web browsers and IDEs.
Service-oriented architecture

SOA enables composition of distributed, heterogeneous web-based services.

SOA requires adherence to certain principles (like stateless services). In many ways a throwback to libraries, but can be very effective.
**Summary**

**Paradigm:** configuration of interacting components

**Motivation:** manage variation
Roadmap

Early history
Objects
Components
**Features**
Metaprogramming
Conclusions
Mixins were introduced in “Flavors”, an early Lisp dialect. A mixin is a fragment of a class that can be mixed in to an existing class to add new features. Mixins avoid the need for multiple inheritance or code duplication in single-inheritance systems. The key drawback is that systems that make heavy use of mixins cannot easily be modified since a change to a mixin can percolate to many classes.
Traits

Class = superclass + state + traits + glue

Traits provide and require methods

<table>
<thead>
<tr>
<th>Trait</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>radius</td>
</tr>
<tr>
<td>bounds</td>
<td>radius:</td>
</tr>
<tr>
<td>diameter</td>
<td>center</td>
</tr>
<tr>
<td>hash</td>
<td>center:</td>
</tr>
</tbody>
</table>

The composing class retains control

Unlike mixins, traits are insensitive to the order of composition. Glue takes the form of explicit aliasing and exclusion of features. Traits are purely static and can be flattened away. “Talents” are dynamic traits for objects.

Stéphane Ducasse, et al. Traits: A Mechanism for fine-grained Reuse. ACM TOPLAS, 2006
Feature-Oriented Programming

FOP is a SPL paradigm for synthesizing programs from features.

Transformations are used to add features to base programs.

The diagram is a “feature model”, or “feature diagram”.

Don Batory and Sean O’Malley. The Design and Implementation of Hierarchical Software Systems With Reusable Components. ACM TOSEM, 1992
Template meta-programming can be used to select features at compile-time. C++ templates are actually Turing-complete. In this example, a factorial is computed at compile-time by composing C++ templates.

Subject-Oriented Programming

SOP adds a “third dimension” to method dispatch.

Procedural invocation is single dispatch; OOP is doubly-dispatched, since it takes the receiver of the message into account; SOP is triply dispatched, by taking the sender into account.

COP offers multi-dimensional dispatch, with multiple “layers” triggered by contextual information.

Each layer may define a number of class extensions (additions, modifications).
**Summary**

*Paradigm:* model and compose elementary features

*Motivation:* features represent domain concepts
Roadmap

Early history

Objects

Components

Features

Metaprogramming

Conclusions
Reflection

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution.


A metaprogram is a program that manipulates a program (possibly itself)
Reflection and reification

**Intercession** is the ability for a program to modify its own execution state or *alter its own interpretation* or meaning.

**Introspection** is the ability for a program to *observe* and therefore reason about its own state.

“A system having itself as application domain and that is causally connected with this domain can be qualified as a reflective system” — Maes, OOPSLA 1987.

NB: Java “reflection” is actually just intercession.
Structural and behavioral reflection

Structural reflection lets you reflect on the *program* being executed

Behavioral reflection lets you reflect on the language *semantics* and *implementation*

Behavioural reflection is especially interesting for realizing language extensions.

Malenfant et al., *A Tutorial on Behavioral Reflection and its Implementation*, 1996
Applications of metaprogramming

**IDE tools**
- debugging
- profiling
- refactoring

**Dynamic applications**
- UI generation
- Modeling tools

Mostly compiler and development tools can benefit from metaprogramming, but some highly dynamic applications make use of it too.
Three approaches to reflection

1. Towers of interpreters are reified on need in practice
2. Reflective languages like Smalltalk are often written in themselves
3. Open implementations like CLOS offer an API (MOP) to the implementation
AOP improves modularity by supporting the separation of cross-cutting concerns.

An aspect packages cross-cutting concerns.

A pointcut specifies a set of join points in the target system to be affected.

“Weaving” is the process of applying the aspect to the target system.


factor out cross-cutting concerns
pointcuts apply aspects to joinpoints in code
joinpoints may be static or dynamic
This (rather lame) foobar example shows how pointcuts can be used to specify not only static locations in the code, but also dynamic locations (ie within the execution of Demo.go()).
Dependency injection externalizes dependencies to be configurable.

**Typical application:** injecting mocks

**Key techniques:**
- parameterization
- code generation
- reflection

Dependency injection refers to techniques to make internal dependencies externally configurable. A good example is to inject mock objects for testing purposes.

MDE makes sense especially when you have the same application running on many platforms.
Slide courtesy Colin Atkinson, Universität Mannheim
The OMG / MDA Stack

The MOF

The UML metamodel ++

Some UML Models ++

Various usages of these models

"the real world"

The OMG / MDA Stack

Slide courtesy Jean Bézivin, INRIA/U Nantes
**Summary**

**Paradigm:** composition as metaprogramming

**Motivation:** separation of base and meta-levels
Roadmap

Early history

Objects

Components

Features

Metaprogramming

Conclusions
Mechanisms

- Call
- Return
- Invocation
- Messages
- Binding
- Code generation
The core composition paradigm is component-based, but it varies along several dimensions.
Conclusion: Trends

---|---|---|---|---|---
procedural technology | object technology | component technology | model technology
Procedures, Pascal, C, ... | Objects, Classes, Smalltalk, C++, ... | Components, Frameworks, Patterns, ... | Models, Transformations, UML, MOF, QVT, ...
procedural refinement | object composition | component composition | model transformation
Attribution-ShareAlike 2.5

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/2.5/