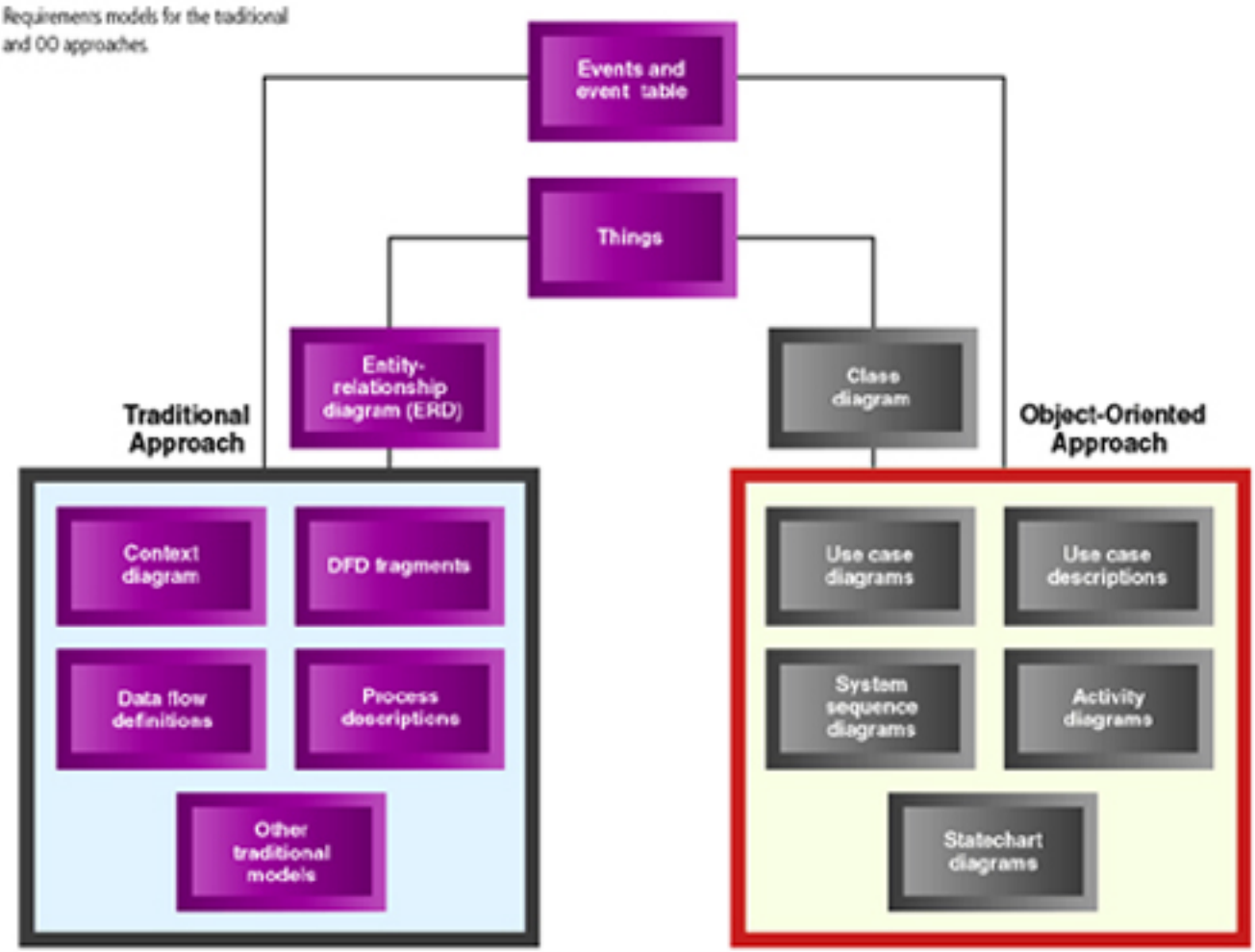
Object-Oriented Programming & Service-Oriented Business Computing



Ligia Derrick
Tia Burnside

First Edition, 2012

ISBN 978-81-323-1289-5

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Object-Oriented Programming

Object-oriented programming (**OOP**) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP.

Overview

An object is a discrete bundle of functions and procedures, often relating to a particular real-world concept such as a bank account holder, a hockey player, or a bulldozer. Other pieces of software can access the object only by calling its functions and procedures that have been allowed to be called by outsiders. A large number of software engineers agree that isolating objects in this way makes their software easier to manage and keep track of. However, a significant number of engineers feel the reverse may be true: that software becomes more complex to maintain and document, or even to engineer from the start. The conditions under which OOP prevails over alternative techniques (and vice-versa) often remain unstated by either party, however, making rational discussion of the topic difficult, and often leading to heated debates over the matter.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming that had been dominant for many years that focused on the *function* of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines). This more conventional approach, which still persists, tends to consider data and behavior separately.

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to 'carry their own operators around with them' (or at least "inherit" them from a similar object or class). In the conventional model, the data and operations on the data don't have a tight, formal association.

History

The terms "objects" and "oriented" in something like the modern sense of object-oriented programming seem to make their first appearance at MIT in the late 1950s and early 1960s. In the environment of the artificial intelligence group, as early as 1960, "object" could refer to identified items (LISP atoms) with properties (attributes); Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966. Another early MIT example was Sketchpad created by Ivan Sutherland in 1960-61; in the glossary of the 1963 technical report based on his dissertation about Sketchpad, Sutherland defined notions of "object" and "instance" (with the class concept covered by "master" or "definition"), albeit specialized to graphical interaction. Also, an MIT ALGOL version, AED-0, linked data structures ("plexes", in that dialect) directly with procedures, prefiguring what were later termed "messages", "methods" and "member functions".

Objects as a formal concept in programming were introduced in the 1960s in Simula 67, a major revision of Simula I, a programming language designed for discrete event simulation, created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo. Simula 67 was influenced by SIMSCRIPT and Hoare's proposed "record classes". Simula introduced the notion of classes and instances or objects (as well as subclasses, virtual methods, coroutines, and discrete event simulation) as part of an explicit programming paradigm. The language also used automatic garbage collection that had been invented earlier for the functional programming language Lisp. Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. The ideas of Simula 67 influenced many later languages, including Smalltalk, derivatives of LISP (CLOS), Object Pascal, and C++.

The Smalltalk language, which was developed at Xerox PARC (by Alan Kay and others) in the 1970s, introduced the term *object-oriented programming* to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67. Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of *Byte magazine*.

In the 1970s, Kay's Smalltalk work had influenced the Lisp community to incorporate object-based techniques that were introduced to developers via the Lisp machine. Experimentation with various extensions to Lisp (like LOOPS and Flavors introducing multiple inheritance and mixins), eventually led to the Common Lisp Object System (CLOS, a part of the first standardized object-oriented programming language, ANSI Common Lisp), which integrates functional programming and object-oriented programming and allows extension via a Meta-object protocol. In the 1980s, there were a few attempts to design processor architectures that included hardware support for objects in memory but these were not successful. Examples include the Intel iAPX 432 and the Linn Smart Rekursiv.

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990's when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0, C++, and Delphi. Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective-C, an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of event-driven programming (although this concept is not limited to OOP). Some feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.

At ETH Zürich, Niklaus Wirth and his colleagues had also been investigating such topics as data abstraction and modular programming (although this had been in common use in the 1960s or earlier). Modula-2 (1978) included both, and their succeeding design, Oberon, included a distinctive approach to object orientation, classes, and such. The approach is unlike Smalltalk, and very unlike C++.

Object-oriented features have been added to many existing languages during that time, including Ada, BASIC, Fortran, Pascal, and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented yet compatible with procedural methodology, such as Python and Ruby. Probably the most commercially important recent object-oriented languages are Visual Basic.NET (VB.NET) and C#, both designed for Microsoft's .NET platform, and Java, developed by Sun Microsystems. Both frameworks show the benefit of using OOP by creating an abstraction from implementation in their own way. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language. Java runs in a virtual machine, making it possible to run on all different operating systems. VB.NET and C# make use of the Strategy pattern to accomplish cross-language inheritance, whereas Java makes use of the Adapter pattern.

Just as procedural programming led to refinements of techniques such as structured programming, modern object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modeling languages (such as UML).

Formal definition

There have been several attempts at formalizing the concepts used in object-oriented programming. The following concepts and constructs have been used as interpretations of OOP concepts:

- coalgebraic datatypes
- abstract data types (which have existential types) allow the definition of modules but these do not support dynamic dispatch
- recursive types
- records are basis for understanding objects if function literals can be stored in fields (like in functional programming languages), but the actual calculi need be considerably more complex to incorporate essential features of OOP. Several extensions of System $F_{<:}$ that deal with mutable objects have been studied; these allow both subtype polymorphism and parametric polymorphism (generics)

Attempts to find a consensus definition or theory behind objects have not proven very successful, and often diverge widely. For example, some definitions focus on mental activities, and some on mere program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping"). OBJECT:=>> Objects are the run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

OOP languages

Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, following distinction can be made:

- Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Smalltalk, Eiffel, Ruby, JADE.
- Languages designed mainly for OO programming, but with some procedural elements. Examples: C++, C#, Java, Python.

- Languages that are historically procedural languages, but have been extended with some OO features. Examples: VB.NET (derived from VB), Fortran 2003, Perl, COBOL 2002, PHP, ABAP.
- Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2).
- Languages with abstract data type support, but not all features of object-orientation, sometimes called object-*based* languages. Examples: Modula-2 (with excellent encapsulation and information hiding), Pliant, CLU.

OOP in scripting

In recent years, object-oriented programming has become especially popular in scripting programming languages. Python, Ruby and Groovy are scripting languages built on OOP principles, while Perl and PHP have been adding object oriented features since Perl 5 and PHP 4, and ColdFusion since version 6.

The Document Object Model of HTML, XHTML, and XML documents on the Internet have bindings to the popular JavaScript/ECMAScript language. JavaScript is perhaps the best known prototype-based programming language, which employs cloning from prototypes rather than inheriting from a class. Another scripting language that takes this approach is Lua. Earlier versions of ActionScript (a partial superset of the ECMA-262 R3, otherwise known as ECMAScript) also used a prototype-based object model. Later versions of ActionScript incorporate a combination of classification and prototype-based object models based largely on the currently incomplete ECMA-262 R4 specification, which has its roots in an early JavaScript 2 Proposal. Microsoft's JScript.NET also includes a mash-up of object models based on the same proposal, and is also a superset of the ECMA-262 R3 specification.

Design patterns

Challenges of object-oriented design are addressed by several methodologies. Most common is known as the design patterns codified by Gamma *et al.*. More broadly, the term "design patterns" can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

Inheritance and behavioral subtyping

It is intuitive to assume that inheritance creates a semantic "is a" relationship, and thus to infer that objects instantiated from subclasses can always be *safely* used instead of those instantiated from the superclass. This intuition is unfortunately false in most OOP languages, in particular in all those that allow mutable objects. Subtype polymorphism as enforced by the type checker in OOP languages (with mutable objects) cannot guarantee behavioral subtyping in any context. Behavioral subtyping is undecidable in general, so it cannot be implemented by a program (compiler). Class or object hierarchies need to be

carefully designed considering possible incorrect uses that cannot be detected syntactically. This issue is known as the Liskov substitution principle.

Matching real world

OOP can be used to translate from real-world phenomena to program elements (and vice versa). OOP was even invented for the purpose of physical modeling in the Simula 67 language. However, not everyone agrees that direct real-world mapping is facilitated by OOP, or is even a worthy goal; Bertrand Meyer argues in *Object-Oriented Software Construction* that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed". At the same time, some principal limitations of OOP had been noted. An example for a real world problem that cannot be modeled elegantly with OOP techniques is the Circle-ellipse problem.

However, Niklaus Wirth (who popularized the adage now known as Wirth's law: "Software is getting slower more rapidly than hardware becomes faster") said of OOP in his paper, "Good Ideas through the Looking Glass", "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours" (contrast KISS principle).

However, it was also noted (e.g. in Steve Yegge's essay *Execution in the Kingdom of Nouns*) that the OOP approach of strictly prioritizing *things* (objects/nouns) before *actions* (methods/verbs) is an paradigma not found in natural languages. This limitation may lead for some real world modelling to overcomplicated results compared e.g. to procedural approaches.

OOP and control flow

OOP was developed to increase the reusability and maintainability of source code. Transparent representation of the control flow had no priority and was meant to be handled by a compiler. With the increasing relevance of parallel hardware and multithreaded coding, developer transparent control flow becomes more important, something hard to achieve with OOP.

Responsibility vs. data-driven design

Responsibility-driven design defines classes in terms of a contract, that is, a class should be defined around a responsibility and the information that it shares. This is contrasted by Wirfs-Brock and Wilkerson with data-driven design, where classes are defined around the data-structures that must be held. The authors hold that responsibility-driven design is preferable.

Performance implications

According to an article in the *Information and Software Technology* journal by Alexander Chatzigeorgiou of the Department of Applied Informatics, at the University of

Macedonia, the object-oriented approach is known to introduce a significant performance penalty compared to classical procedural programming. For instance, profiling results for embedded applications indicate that C++ programs, apart from being slower than their corresponding C versions, consume significantly more energy (mainly due to the increased instruction count, larger code size and increased number of accesses to the data memory for the object-oriented versions).

Why this is so is partly due to the indirect way that objects are modified or examined (by message passing and methods - rather than by direct assignment); partly through the OO requirement to frequently *physically* copy data (as parameters lists into messages, rather than merely pointing to existing data) - resulting in extra memory requirements and its corresponding allocation and deallocation, and partly through the use of dynamic dispatch (as opposed to static calls).

Criticisms

A number of well-known researchers and programmers have criticized OOP. Here is an incomplete list:

- Luca Cardelli wrote a paper titled "Bad Engineering Properties of Object-Oriented Languages".
- Richard Stallman wrote in 1995, "Adding OOP to Emacs is not clearly an improvement; I used OOP when working on the Lisp Machine window systems, and I disagree with the usual view that it is a superior way to program."
- A study by Potok et al. has shown no significant difference in productivity between OOP and procedural approaches.
- Christopher J. Date stated that critical comparison of OOP to other technologies, relational in particular, is difficult because of lack of an agreed-upon and rigorous definition of OOP. Date and Darwen propose a theoretical foundation on OOP that uses OOP as a kind of customizable type system to support RDBMS.
- Alexander Stepanov suggested that OOP provides a mathematically-limited viewpoint and called it "almost as much of a hoax as Artificial Intelligence" (possibly referring to the Artificial Intelligence projects and marketing of the 1980s that are sometimes viewed as overzealous in retrospect).
- Paul Graham has suggested that the purpose of OOP is to act as a "herding mechanism" that keeps mediocre programmers in mediocre organizations from "doing too much damage". This is at the expense of slowing down productive programmers who know how to use more powerful and more compact techniques.
- Joe Armstrong, the principal inventor of Erlang, is quoted as saying "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."
- Richard Mansfield, author and former editor of *COMPUTE!* magazine, states that "like countless other intellectual fads over the years ("relevance", communism, "modernism", and so on—history is littered with them), OOP will be with us until eventually reality asserts itself. But considering how OOP currently pervades both

universities and workplaces, OOP may well prove to be a durable delusion. Entire generations of indoctrinated programmers continue to march out of the academy, committed to OOP and nothing but OOP for the rest of their lives." He also is quoted as saying "OOP is to writing a program, what going through airport security is to flying".

- Rich Hickey, creator of Clojure, described object systems as over simplistic models of the real world. He emphasized the inability of OOP to model time properly, which is getting increasingly problematic as software systems become more concurrent.

Chapter 2

Subtype Polymorphism and This (Computer Science)

Subtype polymorphism

In programming language theory, **subtyping** or **subtype polymorphism** is a form of type polymorphism in which a **subtype** is a datatype that is related to another datatype (the **supertype**) by some notion of substitutability, meaning that program constructs, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be *safely used in a context where* a term of type T is expected. The precise semantics of subtyping crucially depends on the particulars of what "safely used in a context where" means in a given programming language. The type system of a programming language essentially defines its own subtyping relation, which may well be trivial.

Because the subtyping relation allows a term to have (belong to) more than one type, subtyping is a form of type polymorphism, so it is (properly) called subtype polymorphism. In object-oriented programming subtyping is commonly called just *polymorphism*. Subtyping is practically never called this way in type theory or in functional programming, where the unqualified use of "polymorphism" usually refers to parametric polymorphism, as in polymorphic lambda calculus. (Mechanisms similar in purpose, but not identical with parametric polymorphism are known by other names in object-oriented programming, e.g. generics in Java or templates in C++.)

Functional programming languages often allow the subtyping of records. Consequently, simply typed lambda calculus extended with record types is perhaps the simplest theoretical setting in which a useful notion of subtyping may be defined and studied. Because the resulting calculus allows terms to have more than one type, it is no longer a "simple" type theory. Since functional programming languages, by definition, support function literals, which can also be stored in records, records types with subtyping provide some of the features of object-oriented programming. (Unless references are added to the language, record "objects" are immutable). Typically, functional programming languages also provide some, usually restricted, form of parametric polymorphism. In a theoretical setting, it is desirable to study the interaction of the two

features; a common theoretical setting is system $F_{<:}$. Various calculi that attempt to capture the theoretical properties of object-oriented programming may be derived from system $F_{<:}$.

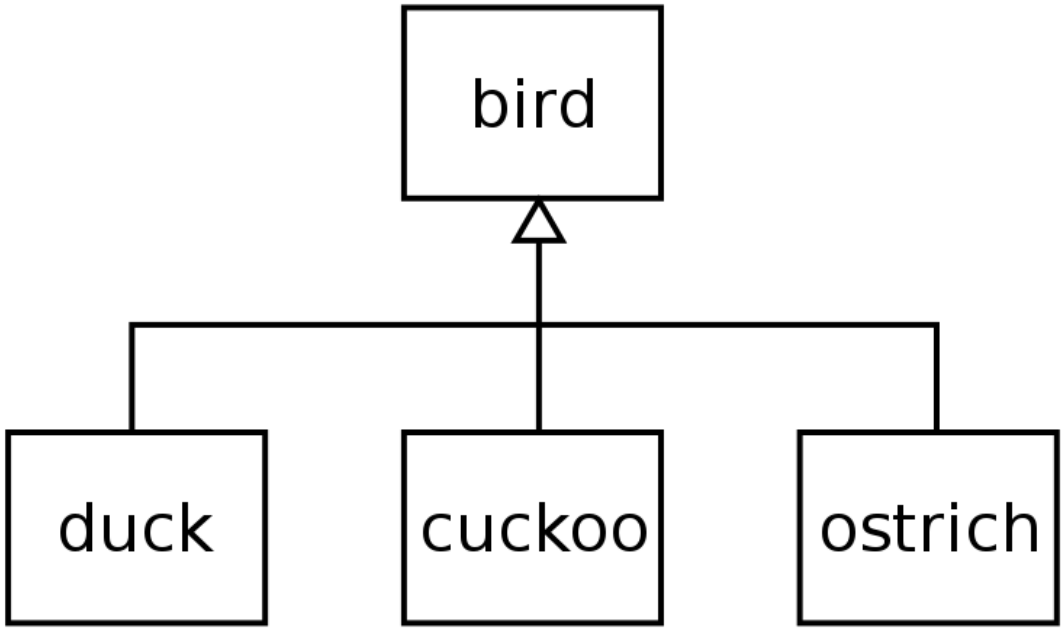
The concept of subtyping is related to the linguistic notions of hypernymy and holonymy. It is also related to the concept of bounded quantification in mathematical logic. Subtyping should not be confused with the notion of (class or object) inheritance from object-oriented languages; subtyping is a relation between types (interfaces in object-oriented parlance) whereas inheritance is a relation between implementations stemming from a language feature that allows new objects to be created from existing ones. In a number of object-oriented languages, subtyping is called **interface inheritance**.

Origins

The notion of subtyping in programming languages dates back to the 1960s; it was introduced in Simula derivatives. The first formal treatments of subtyping were given by John C. Reynolds in 1980 who used category theory to formalize implicit conversions, and Luca Cardelli (1985).

The concept of subtyping has gained visibility (and synonymy with polymorphism in some circles) with the mainstream adoption of object-oriented programming. In this context, the principle of safe substitution is often called the Liskov substitution principle, after Barbara Liskov who popularized it in a keynote address at a conference on object-oriented programming in 1987. Because it must consider mutable objects, the ideal notion of subtyping defined by Liskov and Jeannette Wing, called behavioral subtyping is considerably stronger than what can be implemented in a type checker.

Examples



Example of Subtypes: where bird is the supertype and all others are subtypes as denoted by the arrow in UML notation.

A simple practical example of subtypes is shown in the diagram, right. The type "bird" has three subtypes "duck", "cuckoo" and "ostrich". Conceptually, each of these is a variety of the basic "bird" that inherits many "bird" characteristics but has some specific differences. The UML notation is used in this diagram, with open-headed arrows showing the direction and type of the relationship between the supertype and its subtypes.

As a more practical example, a language might allow floating point values to be used wherever integer values are expected (`Float` <: `Integer`), or it might define a generic type `Number` as a common supertype of integers and the reals. In this second case, we only have `Integer` <: `Number` and `Float` <: `Number`, but `Integer` and `Float` are not subtypes of each other.

Programmers may take advantage of subtyping to write code in a more abstract manner than would be possible without it. Consider the following example:

```
function max (x as Number, y as Number) is
  if x < y then
    return y
  else
    return x
end
```

If integer and real are both subtypes of `Number`, and an operator of comparison with an arbitrary Number is defined for both types, then values of either type can be passed to this function. However, the very possibility of implementing such an operator highly constrains the Number type (for example, one can't compare an integer with a complex number), and actually only comparing integers with integers and reals with reals makes sense. Rewriting this function so that it would only accept 'x' and 'y' of the same type requires F-Bounded Polymorphism.

Subtyping in type theory is characterized by the fact that any expression of type A may also be given type B if $A <: B$; the formal typing rule that codifies this is known as the *subsumption* rule.

Subtyping schemes

Type theorists make a distinction between **nominal subtyping**, in which only types declared in a certain way may be subtypes of each other, and **structural subtyping**, in which the structure of two types determines whether or not one is a subtype of the other. The class-based object-oriented subtyping described above is nominal; a structural subtyping rule for an object-oriented language might say that if objects of type A can handle all of the messages that objects of type B can handle (that is, if they define all the same methods), then A is a subtype of B regardless of whether either inherits from the other. Sound structural subtyping rules for types other than object types are also well known.

Implementations of programming languages with subtyping fall into two general classes: *inclusive* implementations, in which the representation of any value of type A also represents the same value at type B if $A <: B$, and *coercive* implementations, in which a value of type A can be *automatically converted* into one of type B. The subtyping induced by subclassing in an object-oriented language is usually inclusive; subtyping relations that relate integers and floating-point numbers, which are represented differently, are usually coercive.

In almost all type systems that define a subtyping relation, it is reflexive (meaning $A <: A$ for any type A) and transitive (meaning that if $A <: B$ and $B <: C$ then $A <: C$). This makes it a preorder on types.

Record types

Types of records give rise to the concepts of *width* and *depth* subtyping. These express two different ways of obtaining a new type of record that allows the same operations as the original record type.

Recall that a record is a collection of (named) fields. Since a subtype is a type which allows all operations allowed on the original type, a record subtype should support the same operations on the fields as the original type supported.

One kind of way to achieve such support, called *width subtyping*, adds more fields to the record. More formally, every (named) field appearing in the width supertype will appear in the width subtype. Thus, any operation feasible on the supertype will be supported by the subtype.

The second method, called *depth subtyping*, replaces the various fields with their subtypes. That is, the fields of the subtype are subtypes of the fields of the supertype. Since any operation supported for a field in the supertype is supported for its subtype, any operation feasible on the record supertype is supported by the record subtype. Depth subtyping only makes sense for immutable records: for example, you can assign 1.5 to the 'x' field of a real point (a record with two real fields), but you can't do the same to the 'x' field of an integer point (which, however, is a deep subtype of the real point type) because 1.5 is not an integer.

Subyping of records can be defined in System $F_{<:}$, which combines parametric polymorphism with subtyping of record types and is a theoretical basis for many functional programming languages that support both features.

Some systems also support subtyping of labeled disjoint union types (such as algebraic data types). The rule for width subtyping is reversed: every tag appearing in the width subtype must appear in the width supertype.

Function types

If $T_1 \to T_2$ is a function type then a subtype of it is any function $S_1 \to S_2$ with the property that $T_1 <: S_1$ and $S_2 <: T_2$. The argument type of $S_1 \to S_2$ is said to be contravariant because the subtyping relation is reversed for it, whereas the return type is covariant. (Informally, this reversal occurs because the refined type is "more liberal" in the types it accepts and "more conservative" in the type it returns.)

In languages that allow side effects, like most object-oriented languages, subtyping is generally not sufficient to guarantee that a function can be safely used in the context of another. Liskov's work in this area focused on behavioral subtyping, which besides the type system safety discussed here also requires that subtypes preserve all invariants guaranteed by the supertypes in some contract. This definition of subtyping is generally undecidable, so it cannot be verified by a type checker.

The subtyping of mutable references is similar to the treatment of function arguments and return values. Write-only references (or *sinks*) are contravariant, like function arguments; read-only references (or *sources*) are covariant, like return values. Mutable references which act as both sources and sinks are invariant.

Object types

Coercions

In coercive subtyping systems, subtypes are defined by implicit type conversion functions from subtype to supertype. For each subtyping relationship ($S <: T$), a coercion function *coerce*: $S \to T$ is provided, and any object s of type S is regarded as the object $coerce_{S \to T}(s)$ of type T. A coercion function may be defined by composition: if $S <: T$ and $T <: U$ then s may be regarded as an object of type u under the compound coercion ($coerce_{T \to U} \circ coerce_{S \to T}$). The type coercion from a type to itself $coerce_{T \to T}$ is the identity function id_T

Coercion functions for records and disjoint union subtypes may be defined componentwise; in the case of width-extended records, type coercion simply discards any components which are not defined in the supertype. The type coercion for function types may be given by $f'(s) = coerce_{S2 \to T2}(f(coerce_{T1 \to S1}(t)))$, reflecting the contravariance of function arguments and covariance of return values.

The coercion function is uniquely determined given the subtype and supertype. Thus, when multiple subtyping relationships are defined, one must be careful to guarantee that all type coercions are coherent. For instance, if an integer such as 2 : *int* can be coerced to a floating point number (say, 2.0 : *float*), then it is not admissible to coerce 2.1 : *float* to 2 : *int*, because the compound coercion $coerce_{float \to float}$ given by $coerce_{int \to float} \circ coerce_{float \to int}$ would then be distinct from the identity coercion id_{float}.

Intersection and union types

Union types should not to be confused with sum types, the anonymous version of which are called a "unions" in many imperative programming languages.

this (computer science)

In many object-oriented programming languages, **`this`** (also called **`self`** or **`Me`**) is a keyword that is used in instance methods to refer to the object on which they are working. C++, and languages which derive in style from it (such as Java, C#, and PHP) generally use `this`. Smalltalk and others such as Object Pascal, Python, Ruby, and Objective-C use `self`; Visual Basic uses `Me`.

The concept is the same in all languages. For brevity, here we just say `this`.

`this` is usually an immutable reference or pointer which refers to the current object. Some languages, such as Objective-C, allow assignment to `this`, although it is

deprecated. Doing so can be very misleading to maintenance programmers, because the assignment does not modify the original object, only changing which object that the rest of the code in the method refers to, and can end with undefined behavior.

After an object is properly constructed `this` is always a valid reference. Some languages require it explicitly; others use lexical scoping to use it implicitly to bring symbols within their class visible. In the latter case, the use of `this` in code, while not illegal, may raise warning bells to a maintenance programmer; although there are still legitimate uses of `this` in this case, such as referring to instance variables hidden by local variables of the same name, or if the method wants to return a reference to the current object, i.e. `this`, itself.

`this` becomes an extra parameter to an instance method. For example, the following instance method in C++

```
int foo::print (bar x)
```

is essentially equivalent to the procedural programming:

```
int foo_print (foo *const this, bar x)
```

In some languages, for example Python and Perl 5, `this` is made explicit, the first parameter of an instance method being such a reference. It needs explicitly to be specified. In this case, the parameter need not necessarily be named `this` or `self`; like any other parameter, it can be freely named by the programmer; however, by informal convention, the first parameter of an instance method in Perl and Python is named `self`.

In some compilers (for example GCC), pointers to C++ instance methods can be directly cast to a pointer of another type, with an explicit `this` pointer parameter.

Static methods in C++ or Java are not associated with instances but classes, and so cannot use `this`, because there is no object. In others, such as Python, Ruby, or Smalltalk, the method is associated with a *class object* that is passed as `this`, and are called class methods.

Implementations

C++

Early versions of C++ would let the `this` pointer be changed; by doing so a programmer could change which object a method was working on. This feature was eventually deprecated, and now `this` in C++ is const .

Early versions of C++ did not include references and it has been suggested that had they been so in C++ from the beginning, `this` would have been a reference, not a pointer. But on the other hand, others say that `this` being a pointer makes life simpler because it

avoids complications that could arise from the class having its address-of operator overloaded. This mostly comes down to troubles with the type system, something that in C++ is well-defined but rather complex.

C++ lets objects destroy themselves with the idiom `delete this`. If done it invalidates the object and the `this` pointer.

Java

A Java language keyword that represents the current instance of the class in which it appears. It is used to access class variables and methods.

Since all instance methods are virtual in Java, `this` can never be null.

"...the reason for using this construct [this] is that we have a situation that is known as ***name overloading*** *- the same name being used for two different entities....It is important to understand that the fields and the parameters are separate variables that exist independently of each other, even though they share similar names. A parameter and a field sharing a name is not really a problem in Java."* - Objects First with Java, D. Barnes and M. Kölling

C#

`this` in C# works the same way as in Java, for reference types. However, within C# "value types", `this` has quite different semantics, being similar to an ordinary mutable variable reference, and can even occur on the left side of an assignment.

Dylan

In Dylan, an OO language that supports multimethods and hasn't a concept of `this`, sending a message to an object is still kept in the syntax. The two forms below work the same; the differences are just syntactic sugar.

```
object.method(param1, param2)
```

and

```
method (object, param1, param2)
```

Python

In Python, there is no keyword for `this`, but it exists as the name of the obligatory first argument of all member functions. Conventionally, the name `self` is used.

Self

The Self language is named after this use of "self".

Xbase++

`Self` is strictly used within methods of a class. Another way to refer to `Self` is to use `::`.

Chapter 3

Class and Instance (Computer Science)

Class (computer science)

In object-oriented programming, a **class** is a construct that is used as a blueprint (or template) to create objects of that class. This blueprint describes the state and behavior that the objects of the class all share. An object of a given class is called an instance of the class. The class that contains (and was used to create) that instance can be considered as the type of that object, e.g. an object instance of the "Fruit" class would be of the type "Fruit".

A class usually represents a noun, such as a person, place or (possibly quite abstract) thing - it is a model of a concept within a computer program. Fundamentally, it encapsulates the *state* and *behavior* of the concept it represents. It encapsulates *state* through data placeholders called attributes (or *member variables* or *instance variables*); it encapsulates *behavior* through reusable sections of code called *methods*.

More technically, a class is a cohesive package that consists of a particular kind of metadata. A class has both an interface and a structure. The interface describes how to interact with the class and its instances using methods, while the structure describes how the data is partitioned into attributes within an instance. A class may also have a representation (metaobject) at run time, which provides run time support for manipulating the class-related metadata. In object-oriented design, a class is the most specific type of an object in relation to a specific layer.

Programming languages that support classes subtly differ in their support for various class-related features. Most support various forms of class inheritance. Many languages also support features providing encapsulation, such as access specifiers.

Reasons for using classes

Computer programs usually model aspects of some real or abstract world (the Domain). Because each class models a concept, classes provide a more natural way to create such models. Each class in the model represents a noun in the domain, and the methods of the class represent verbs that may apply to that noun. For example in a typical business system, various aspects of the business are modeled, using such classes as *Customer*,

Product, *Worker*, *Invoice*, *Job*, etc. An *Invoice* may have methods like *Create*, *Print* or *Send*, a *Job* may be *Performed* or *Canceled*, etc. Once the system can model aspects of the business accurately, it can provide users of the system with useful information about those aspects. Classes allow a clear correspondence (mapping) between the model and the domain, making it easier to design, build, modify and understand these models. Classes provide some control over the often challenging complexity of such models.

Classes can accelerate development by reducing redundant program code, testing and bug fixing. If a class has been thoroughly tested and is known to be a 'solid work', it is usually true that using or extending the well-tested class will reduce the number of bugs - as compared to the use of freshly-developed or ad hoc code - in the final output. In addition, efficient class reuse means that many bugs need to be fixed in only one place when problems are discovered.

Another reason for using classes is to simplify the relationships of interrelated data. Rather than writing code to repeatedly call a graphical user interface (GUI) window drawing subroutine on the terminal screen (as would be typical for structured programming), it is more intuitive. With classes, GUI items that are similar to windows (such as dialog boxes) can simply inherit most of their functionality and data structures from the window class. The programmer then need only add code to the dialog class that is unique to its operation. Indeed, GUIs are a very common and useful application of classes, and GUI programming is generally much easier with a good class framework.

Instantiation

A class is used to create new **instances** (*objects*) by **instantiating** the class.

Instances of a class share the same set of attributes, yet may differ in what those attributes contain. For example, a class "Person" would describe the attributes common to all instances of the Person class. Each person is generally like the others, but they vary in such attributes as "height" and "weight". The description of the class would name such attributes and define the actions which a person can perform, such as "run", "jump", "sleep", "walk", etc.

Class also refers to the file that contains modules or other codes in java programming.

Interfaces and methods

Note: the term "interface" here isn't referring to a Java interface, although the two are closely related.

Objects define their interaction with the outside world through the **methods** that they expose. A method, or *instance method*, is a subroutine (function) with a special property that it has access to data stored in an object (instance). Methods that manipulate the data of the object and perform tasks are sometimes described as behavior. All objects belong to a class.

Methods form the object's **interface** with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to toggle the television on and off. In this example, the television is the instance, each method is represented by a button, and all the buttons together comprise the interface. In its most common form, an interface is a specification of a group of related methods without any associated implementation of the methods.

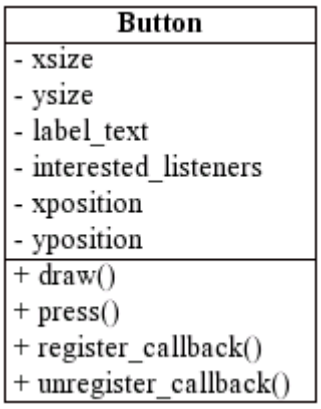
Every class **implements** (or *realizes*) an interface by providing structure (i.e. data and state) and method implementations (i.e. providing code that specifies how methods work). There is a distinction between the definition of an interface and the implementation of that interface. In most languages, this line is usually blurred, because a class declaration both defines and implements an interface. Some languages, however, provide features that help separate interface and implementation. For example, an abstract class can define an interface without providing implementation.

Interfaces may also be defined to include a set of auxiliary functions called *static methods* or *class methods*. Static methods, like instance methods, are exclusively associated with the class. They differ from instance methods in that they do not work with instances of the class; that is, static methods neither require an instance of the class nor can they access the data of such an instance. For example, getting the total number of televisions in existence could be a static method of the television class. This method is clearly associated with the class, yet is outside the domain of each individual instance of the class. Another example is a static method that finds a particular instance out of the set of all television objects.

Languages that support class inheritance also allow classes to inherit interfaces from the classes that they are derived from. In languages that support access specifiers, the interface of a class is considered to be the set of public members of the class, including both methods and attributes (via implicit getter and setter methods); any private members or internal data structures are not intended to be depended on by client code and thus are not part of the interface.

The object-oriented programming methodology is designed in such a way that the operations of any interface of a class are usually chosen to be independent of each other. It results in a client–server (or layered) design where servers do not depend in any way on the clients. An interface places no requirements for clients to invoke the operations of one interface in any particular order. This approach has the benefit that client code can assume that the operations of an interface are available for use whenever the client holds a valid reference to the object.

Structure of a class



UML notation for classes

Along with having an interface, a class contains a description of structure of data stored in the instances of the class. The data is partitioned into **attributes** (or *properties*, *fields*, *data members*). Going back to the television set example, the myriad attributes, such as size and whether it supports color, together comprise its structure. A class represents the full description of a television, including its attributes (structure) and buttons (interface).

The state of an instance's data is stored in some resource, such as memory or a file. The storage is assumed to be located in a specific location, such that it is possible to access the instance through references to the identity of the instances. However, the actual storage location associated with an instance may change with time. In such situations, the identity of the object does not change. The state is encapsulated and every access to the state occurs through methods of the class.

In most languages, the structures as defined by the class determine how the memory used by its instances will be laid out. This technique is known as the *cookie-cutter model*. The alternative to the cookie-cutter model is the model of Python, wherein objects are structured as associative key-value containers. In such models, objects that are instances of the same class could contain different instance variables, as state can be dynamically added to the object. This may resemble prototype-based languages in some ways, but it is not equivalent.

A class also describes a set of invariants that are preserved by every method in the class. An invariant is a constraint on the state of an object that should be satisfied by every object of the class. The main purpose of the invariants is to establish what objects belong to the class. An invariant is what distinguishes data types and classes from each other; that is, a class does not allow use of all possible values for the state of the object, and instead allows only those values that are well-defined by the semantics of the intended use of the data type. The set of supported (public) methods often implicitly establishes an invariant. Some programming languages support specification of invariants as part of the definition of the class, and enforce them through the type system. Encapsulation of state is necessary for being able to enforce the invariants of the class.

Some languages allow an implementation of a class to specify constructor (or *initializer*) and destructor (or *finalizer*) methods that specify how instances of the class are created and destroyed, respectively. A constructor that takes arguments can be used to create an instance from passed-in data. The main purpose of a constructor is to establish the invariant of the class, failing if the invariant isn't valid. The main purpose of a destructor is to destroy the identity of the instance, invalidating any references in the process. Constructors and destructors are often used to reserve and release, respectively, resources associated with the object. In some languages, a destructor can return a value which can then be used to obtain a public representation (transfer encoding) of an instance of a class and simultaneously destroy the copy of the instance stored in current thread's memory.

A class may also contain *static attributes* or *class attributes*, which contain data that are specific to the class yet are common to all instances of the class. If the class itself is treated as an instance of a hypothetical metaclass, static attributes and static methods would be instance attributes and instance methods of that metaclass.

Run-time representation of classes

As a data type, a class is usually considered as a compile-time construct. A language may also support prototype or factory metaobjects that represent run-time information about classes, or even represent metadata that provides access to reflection facilities and ability to manipulate data structure formats at run-time. Many languages distinguish this kind of run-time type information about classes from a class on the basis that the information is not needed at run-time. Some dynamic languages do not make strict distinctions between run-time and compile-time constructs, and therefore may not distinguish between metaobjects and classes.

For example: if Human is a metaobject representing the class Person, then instances of class Person can be created by using the facilities of the Human metaobject.

Information hiding and encapsulation

Many languages support the concept of **information hiding** and **encapsulation**, typically with **access specifiers** for class members. Access specifiers specify constraints on who can access which class members. Some access specifiers may also control how classes inherit such constraints. Their primary purpose is to separate the interface of a class from its implementation.

A common set of access specifiers that many object-oriented languages support is:

- **Private** (or *class-private*) restricts the access to the class itself. Only methods that are part of the same class can access private members.
- **Protected** (or *class-protected*) allows the class itself and all its subclasses to access the member.
- **Public** means that any code can access the member by its name.

Note that although many languages support the above access specifiers, the semantics of them may subtly differ in each.

A common usage of access specifiers is to separate the internal data (structure)s of a class from its interface; that is, the internal data structures are private. Public accessor methods can be used to inspect or alter such private data. The various object-oriented programming languages enforce this to various degrees. For example, the Java language does not allow client code to access the private data of a class at all, whereas in languages like Objective-C or Perl client code can do whatever it wants. In C++ language, private methods are visible but not accessible in the interface; however, they are commonly made invisible by explicitly declaring fully abstract classes that represent the interfaces of the class.

Access specifiers do not necessarily control **visibility**, in that even private members may be visible to client code. In some languages, an inaccessible but visible member may be referred to at run-time (e.g. pointer to it can be returned from member functions), but all attempts to use it by referring to the name of the member from client code will be prevented by the type checker. Object-oriented design uses the access specifiers in conjunction with careful design of public method implementations to enforce class invariants. Access specifiers are intended to protect against accidental use of members by clients, but are not suitable for run-time protection of object's data.

Some languages, such as Ruby, support **instance-private** and **instance-protected** access specifiers in lieu of (or in addition to) class-private and class-protected, respectively. They differ in that they restrict access based on the instance itself, rather than the instance's class.

In addition, some languages, such as C++, support a mechanism where a function explicitly declared as *friend* of the class may access the members designated as private or protected.

Associations between classes

In object-oriented design and in UML, an **association** between two classes is a type of a link between the corresponding objects. A (two-way) association between classes A and B describes a relationship between each object of class A and some objects of class B, and vice versa. Associations are often named with a verb, such as "subscribes-to".

An association role type describes the role type of an instance of a class when the instance participates in an association. An association role type is related to each end of the association. A role describes an instance of a class from the point of view of a situation in which the instance participates in the association. Role types are collections of role (instance)s grouped by their similar properties. For example, a "subscriber" role type describes the property common to instances of the class "Person" when they participate in a "subscribes-to" relationship with the class "Magazine". Also, a "Magazine" has the "subscribed magazine" role type when the subscribers subscribe-to it.

Association role multiplicity describes how many instances correspond to each instance of the other class(es) of the association. Common multiplicities are "0..1", "1..1", "1..*" and "0..*", where the "*" specifies any number of instances.

There are some special kinds of associations between classes.

Composition

Composition between class A and class B describes a **has-a** relationship where instances of class B have shorter or same lifetime than the lifetime of the corresponding instances of the enclosing class. Class B is said to be a part of class A. This is often implemented in programming languages by allocating the data storage of instances of class A to contain a representation of instances of class B.

Aggregation is a variation of composition that describes that instances of a class are part of instances of the other class, but the constraint on lifetime of the instances is not required. The implementation of aggregation is often via a pointer or reference to the contained instance. In both cases, method implementations of the enclosing class can invoke methods of the part class. A common example of aggregation is a list class. When a list's lifetime is over, it does not necessarily mean the lifetimes of the objects within the list are also over.

Inheritance

Another type of class association is **inheritance**, which involves **subclasses** and **superclasses**, also known respectively as *child classes* (or *derived classes*) and *parent classes* (or *base classes*). If [car] was a class, then [station wagon] and [mini-van] might be two subclasses. If [Button] is a subclass of [Control], then all buttons are controls. In other words, **inheritance** is an **is-a** relationship between two classes. Subclasses usually consist of several kinds of modifications (customizations) to their respective superclasses: addition of new instance variables, addition of new methods and overriding of existing methods to support the new instance variables.

Conceptually, a superclass should be considered as a common part of its subclasses. This factoring of commonality is one mechanism for providing reuse. Thus, extending a superclass by modifying the existing class is also likely to narrow its applicability in various situations. In object-oriented design, careful balance between applicability and functionality of superclasses should be considered. Subclassing is different from subtyping in that subtyping deals with common behaviour whereas subclassing is concerned with common structure.

Some programming languages (for example C++) allow multiple inheritance - they allow a child class to have more than one parent class. This technique has been criticized by some for its unnecessary complexity and being difficult to implement efficiently, though some projects have certainly benefited from its use. Java, for example has no multiple

inheritance, as its designers felt that it would add unnecessary complexity. Java instead allows inheriting from multiple pure abstract classes (called interfaces in Java).

Sub- and superclasses are considered to exist within a hierarchy defined by the inheritance relationship. If multiple inheritance is allowed, this hierarchy is a directed acyclic graph (or DAG for short), otherwise it is a tree. The hierarchy has classes as nodes and inheritance relationships as links. The levels of this hierarchy are called layers or levels of abstraction. Classes in the same level are more likely to be associated than classes in different levels.

There are two slightly different points of view as to whether subclasses of the same class are required to be disjoint. Sometimes, subclasses of a particular class are considered to be completely disjoint. That is, every instance of a class has exactly one *most-derived class*, which is a subclass of every class that the instance has. This view does not allow dynamic change of object's class, as objects are assumed to be created with a fixed most-derived class. The basis for not allowing changes to object's class is that the class is a compile-time type, which does not usually change at runtime, and polymorphism is utilized for any dynamic change to the object's behavior, so this ability is not necessary. And design that does not need to perform changes to object's type will be more robust and easy-to-use from the point of view of the users of the class.

From another point of view, subclasses are not required to be disjoint. Then there is no concept of a most-derived class, and all types in the inheritance hierarchy that are types of the instance are considered to be equally types of the instance. This view is based on a dynamic classification of objects, such that an object may change its class at runtime. Then object's class is considered to be its *current* structure, but changes to it are allowed. The basis for allowing changes to object's class is a perceived inconvenience caused by replacing an instance with another instance of a different type, since this would require change of all references to the original instance to be changed to refer to the new instance. When changing the object's class, references to the existing instances do not need to be replaced with references to new instances when the class of the object changes. However, this ability is not readily available in all programming languages. This analysis depends on the proposition that dynamic changes to object structure are common. This may or may not be the case in practice.

Languages without inheritance

Although class-based languages are commonly assumed to support inheritance, inheritance is not an intrinsic aspect of the concept of classes. There are languages that support classes yet do not support inheritance. Examples are earlier versions of Visual Basic. These languages, sometimes called "object-based languages", do not provide the structural benefits of statically type-checked interfaces for objects. This is because in object-based languages, it is possible to use and extend data structures and attach methods to them at run-time. This precludes the compiler or interpreter being able to check the type information specified in the source code as the type is built dynamically

and not defined statically. Most of these languages allow for *instance behaviour* and complex *operational polymorphism*.

Categories of classes

There are many categories of classes depending on modifiers. Note that these categories do not necessarily categorize classes into distinct partitions. For example, while it is impossible to have a class that is both abstract and concrete, a sealed class is implicitly a concrete class, and it may be possible to have an abstract partial class.

Concrete classes

A **concrete class** is a class that can be instantiated. This contrasts with abstract classes as described below.

Abstract classes

An **abstract class**, or *abstract base class* (ABC), is a class that cannot be instantiated. Such a class is only meaningful if the language supports inheritance. An abstract class is designed *only* as a parent class from which child classes may be derived. Abstract classes are often used to represent abstract concepts or entities. The incomplete features of the abstract class are then shared by a group of subclasses which add different variations of the missing pieces.

Abstract classes are superclasses which contain abstract methods and are defined such that concrete subclasses are to extend them by implementing the methods. The behaviors defined by such a class are "generic" and much of the class will be undefined and unimplemented. Before a class derived from an abstract class can become concrete, i.e. a class that can be instantiated, it must implement particular methods for all the abstract methods of its parent classes.

When specifying an abstract class, the programmer is referring to a class which has elements that are meant to be implemented by inheritance. The abstraction of the class methods to be implemented by the subclasses is meant to simplify software development. This also enables the programmer to focus on planning and design.

Most object oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java, the keyword *abstract* is used. In C++, an abstract class is a class having at least one abstract method (a pure virtual function in C++ parlance).

Some languages, notably Java and C#, additionally support a variant of abstract classes called an interface. Such a class can only contain abstract publicly-accessible methods. In this way, they are closely related - but not equivalent - to the abstract concept of interfaces described here.

Sealed classes

Some languages also support **sealed classes**. A sealed class cannot be used as a base class. For this reason, it also cannot be an abstract class. Sealed classes are primarily used to prevent derivation. They add another level of strictness during compile-time, improve memory usage, and trigger certain optimizations that improve run-time efficiency.

Local and inner classes

In some languages, classes can be declared in scopes other than the global scope. There are various types of such classes.

One common type is an **inner class**, which is a class defined within another class. Since it involves two classes, this can also be treated as another type of class association. The methods of an inner class can access static methods of the enclosing class(es). An inner class is typically not associated with instances of the enclosing class, i.e. an inner class is not instantiated along with its enclosing class. Depending on language, it may or may not be possible to refer to the class from outside the enclosing class. A related concept is *inner types* (a.k.a. *inner data type*, *nested type*), which is a generalization of the concept of inner classes. C++ is an example of a language that supports both inner classes and inner types (via *typedef* declarations).

Another type is a **local class**, which is a class defined within a procedure or function. This limits references to the class name to within the scope where the class is declared. Depending on the semantic rules of the language, there may be additional restrictions on local classes compared non-local ones. One common restriction is to disallow local class methods to access local variables of the enclosing function. For example, in C++, a local class may refer to static variables declared within its enclosing function, but may not access the function's automatic variables.

Named vs. anonymous classes

In most languages, a class is bound to a name or identifier upon definition. However, some languages allow classes to be defined without names. Such a class is called an **anonymous class** (analogous to named vs. anonymous functions).

Metaclasses

Metaclasses are classes whose instances are classes. A metaclass describes a common structure of a collection of classes. A metaclass can implement a design pattern or describe a shorthand for particular kinds of classes. Metaclasses are often used to describe frameworks.

In some languages such as Python, Ruby, Java, and Smalltalk, a class is also an object; thus each class is an instance of the unique metaclass, which is built in the language. For example, in Objective-C, each object and class is an instance of NSObject. The Common

Lisp Object System (CLOS) provides metaobject protocols (MOPs) to implement those classes and metaclasses.

Partial classes

Partial classes are classes that can be split over multiple definitions (typically over multiple files), making it easier to deal with large quantities of code. At compile time the partial classes are grouped together, thus logically make no difference to the output. An example of the use of partial classes may be the separation of user interface logic and processing logic. A primary benefit of partial classes is allowing different programmers to work on different parts of the same class at the same time. They also make automatically generated code easier to interpret, as it is separated from other code into a partial class.

Partial classes have existed in Smalltalk under the name of *Class Extensions* for considerable time. With the arrival of the .NET framework 2, Microsoft introduced partial classes, supported in both C# 2.0 and Visual Basic 2005.

Other Approaches

To the surprise of some familiar with the use of classes, classes are not the only way to approach object-oriented programming. Another common approach is prototype-based programming. Languages that support non-class-based programming are usually designed with the motive to address the problem of tight-coupling between implementations and interfaces due to the use of classes. For example, the Self language, a prototype-based language, was designed to show that the role of a class can be substituted by using an extant object which serves as a prototype to a new object, and the resulting language is as expressive as Smalltalk with more generality in creating objects.

Examples

Java

```
public class Example
{
    public static void main (String args[])
    {
        System.out.println("Hello World");
    }
}
```

PHP

```
class DateObject {
        public function getTime() {
                return(time());
        }
```

```
        public function getDate() {
                return(date('jS F, Y', $this->getTime());
        }
}
```

Ruby

```
class Person
    def initialize(name)
        @name = name
    end
    def print_name
        print "Hello " + @name + "!"
    end
end

person = Person.new("Jane Doe")
person.print_name
```

Instance (computer science)

In object-oriented programming an instance is an occurrence or a copy of an object, whether currently executing or not. Instances of a class share the same set of attributes, yet will typically differ in what those attributes contain. For example, a class "Employee" would describe the attributes common to all instances of the Employee class. For the purposes of the task being solved Employee objects may be generally alike, but vary in such attributes as "name" and "salary". The description of the class would itemize such attributes and define the operations or actions relevant for the class, such as "increase salary" or "change telephone number". One could then talk about one instance of the Employee object with name = "Jane Doe" and another instance of the Employee object with name = "John Doe".

Chapter 4

Method (Computer Science) and Message Passing

Method (computer science)

In object-oriented programming, a **method** is a subroutine that is exclusively associated either with a class (in which case it is called a *class method* or a *static method*) or with an object (in which case it is an *instance method*). Like a subroutine in procedural programming languages, a method usually consists of a sequence of programming statements to perform an action, a set of input parameters to customize those actions, and possibly an output value (called the *return value*) of some kind. Methods provide a mechanism for accessing and manipulating the encapsulated data stored in an object.

Types of methods

As mentioned above, *instance* methods are associated with an object, while *class* or *static* methods are associated with a class. In the object oriented programming paradigm, for each and every means of access to the underlying data (from its creation, initialization, retrieval and modification to its ultimate "destruction", (indirect) methods are intentionally used in preference to direct assignments.

- An *Instance* method, in a typical implementation, is passed a hidden reference (e.g. `this`, `self` or `Me`) to the object (whether a class or class instance) it belongs to, so that it can access the data associated with it.

  For *Class/static* methods, this may or may not happen according to the language. A typical example of a class method would be one that keeps count of the number of created objects within a given class.

- An *abstract* method is a dummy code method which has no implementation. It is often used as a placeholder to be overridden later by a subclass of or an object prototyped from the one that implements the abstract method. In this way, abstract methods help to partially specify a framework.

- An *accessor* method is a method that is usually small, simple and provides the sole means for the state of an object to be accessed (retrieved) from other parts of a program.

  Although this introduces a new dependency, as stated above, use of methods are preferred, in the object oriented paradigm, to directly accessing state data - because they provide an abstraction layer. For example, if a bank-account class provides a `getBalance()` accessor method to retrieve the current balance (rather than directly accessing the balance data fields), then later revisions of the same code can implement a more complex mechanism for balance retrieval (say, a database fetch), without the dependent code needing to be changed (However, this often claimed advantage is not unique to object oriented programming, and was earlier implemented - when desirable in critical systems - through conventional modular programming with optional run-time, systemwide locking mechanisms, in the imperative/procedural paradigms)
  To compare the value of two data items, two accessor method calls are normally required before a comparison can take place between the retrieved primitive data type values. *Comparator* methods are required to compare entire objects for equality. This contrasts with a direct comparison in non-OOP paradigms.

- An *update*, *modifier*, or *mutator* method, is an accessor method that changes the state of an object. Objects that provide such methods are considered mutable objects.
- A *constructor* method, supported by many languages, is called automatically upon the creation of an instance of a class. Some languages have a special syntax for constructors.

  - In Java, C++, C#, ActionScript, and PHP constructors have the same name as the class of which they are a member (PHP 5 also allows `__construct` as a constructor)
  - In Perl constructors are, by convention, named *new* and have to do a fair amount of object creation.
  - In Moose object system for Perl, constructors (named *new*) are automatically created and are extended by specifying a *BUILD* method.
  - In Visual Basic .NET, the constructor is called "`New`".
  - In Python, the constructor is called `__init__` and is always passed its parent class as an argument, the name for which is generally defined as `self`.
  - Object Pascal constructors are signified by the keyword `constructor` and can have user-defined names (but are mostly called "`Create`").
  - In Objective-C the constructor method is split across two methods, `alloc` and `init`, with the `alloc` method setting aside memory for an instance of the class and the `init` method handling the bulk of initializing the instance. A call to the `new` method invokes both the `alloc` and the `init` method for the class instance.

- A destructor method (i.e. a special instance method that is called automatically upon the destruction of an instance of a class), is implemented in some languages.
    - In C++, destructors are distinguished by having the same name as the class of the object they're associated with, but with the addition of a tilde (~) in front (or `__destruct` in PHP 5).
    - In Object Pascal, destructors have the keyword `destructor` and can have user-defined names (but are mostly called "Destroy").
    - In Perl, the destructor method is named *DESTROY*.
    - In Moose object system for Perl, the destrctor method is named *DEMOLISH*.
    - In Objective-C the destructor method is named `dealloc`.

Isolation levels

Whereas a C programmer might push a value onto a stack data-structure by calling:

```
stackPush (&myStack, value);
```

a C++ programmer would write:

```
myStack.push(value);
```

The difference is the required level of isolation. In C, the `stackPush` procedure could be in the same source file as the rest of the program and if it was, any other pieces of the program in that source file could see and modify all of the low level details of how the stack was implemented, completely bypassing the intended interface. In C++, regardless of where the class is placed, only the methods which are part of `myStack` will be able to get access to those low-level details without going through the formal interface methods. Languages such as C can provide comparable levels of protection by using different source files and not providing external linkage to the private parts of the stack implementation.

Design intent

According to the design by contract methodology, a public method should preserve the class invariants of the object it is associated with, and should always assume that they are valid when it commences execution (private methods do not necessarily need to follow this recommendation). To this effect, preconditions are used to constrain the method's parameters, and postconditions to constrain method's output, if it has one. If any one of either the preconditions or postconditions is not met, a method may raise an exception. If the object's state does not satisfy its class invariants on entry to or exit from any method, the program is considered to have a bug.

The difference between procedures in general and an object's method is that the latter, being associated with a particular object, may access or modify the data private to that

object in a way consistent with the intended behavior of the object. Consequently, rather than thinking "a method is just a sequence of commands", a programmer using an object-oriented language will consider a method to be "an object's way of providing a service" (its "method of doing the job", hence the name); a method call is thus considered to be a *request* to an object to perform some task.

Consequently, method calls are often modelled as a means of passing a message to an object. Rather than directly performing an operation on an object, a message (most of the time accompanied by parameters) is sent to the object telling it what it should do. The object either complies or raises an exception describing why it cannot do so. Applied to our stack example, rather than pushing a value onto the stack, a value is sent to the stack, along with the message "push".

Static methods

As mentioned above, a method may be declared as static, meaning that it acts at the class level rather than at the instance level. Therefore, a static method cannot refer to a specific instance of the class (i.e. it cannot refer to `this`, `self`, `Me`, etc.), unless such references are made through a parameter referencing an instance of the class, although in such cases they must be accessed through the parameter's identifier instead of `this`. Most importantly there is no need to make an object for accessing data .i.e. without creating an object we can access the data members of a static class. An example of a static member and its consumption in C# code:

```
public class ExampleClass
{
  public static void StaticExample()
  {
     // static method code
  }

  public void InstanceExample()
  {
     // instance method code here
     // can use THIS
  }
}

/// Consumer of the above class:

// Static method is called -- no instance is involved
ExampleClass.StaticExample();

// Instance method is called
ExampleClass objMyExample = new ExampleClass();
objMyExample.InstanceExample();
```

Confusingly, methods marked as `class` in Object Pascal also cannot refer to a class object, as can class methods in Python or Smalltalk. For example, this Python method can

create an instance of `Dict` or of any subclass of it, because it receives a reference to a *class object* as `cls`:

```
class Dict(object):
   @classmethod
   def fromkeys (cls, iterable, value=None):
       d = cls()
       for key in iterable:
           d[key] = value
       return d
```

Message passing

Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Overview

Message passing is the paradigm of communication where messages are sent from a sender to one or more recipients. Forms of messages include (remote) method invocation, signals, and data packets. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

Prominent theoretical foundations of concurrent computation, such as the Actor model and the process calculi are based on message passing. Implementations of concurrent systems that use message passing can either have message passing as an integral part of the language, or as a series of library calls from the language. Examples of the former include many distributed object systems. Examples of the latter include Microkernel operating systems pass messages between one kernel and one or more server blocks, and the Message Passing Interface used in High Performance Computing.

Message passing systems and models

Distributed object and remote method invocation systems like ONC RPC, Corba, Java RMI, DCOM, SOAP, .NET Remoting, CTOS, QNX Neutrino RTOS, OpenBinder, D-Bus and similar are message passing systems.

Message passing systems have been called "shared nothing" systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor, process, thread, socket, *etc.*). Such messaging is used in Web Services by SOAP. This concept is the higher-level version of a datagram except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.

Messages are also commonly used in the same sense as a means of interprocess communication; the other common technique being streams or pipes, in which data are sent as a sequence of elementary data items instead (the higher-level version of a virtual circuit).

Synchronous versus Asynchronous Message Passing

Synchronous message passing systems require the sender and receiver to wait for each other to transfer the message. That is, the sender will not continue until the receiver has received the message.

Synchronous communication has two advantages. The first advantage is that reasoning about the program can be simplified in that there is a synchronisation point between sender and receiver on message transfer. The second advantage is that no buffering is required. The message can always be stored on the receiving side, because the sender will not continue until the receiver is ready.

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready. The advantage of asynchronous communication is that the sender and receiver can overlap their computation because they do not wait for each other.

Synchronous communication can be built on top of asynchronous communication by ensuring that the sender always wait for an acknowledgement message from the receiver before continuing.

The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future

messages. If the sender is blocked, it may lead to an unexpected deadlock. If messages are dropped, then communication is no longer reliable.

Message passing versus Calling

Message passing should be contrasted with the alternative communication method for passing information between programs - the Call. In a traditional `Call`, arguments are passed to the "callee" (the receiver) typically by one or more general purpose registers or in a parameter list containing the addresses of each of the arguments. This form of communication differs from message passing in at least three crucial areas -

- total memory usage
- transfer time
- locality

In message passing, each of the arguments has to have sufficient available *extra* memory for copying the existing argument into a portion of the new message. This applies irrespective of the size of the original arguments - so if one of the arguments is (say) an HTML string of 31,000 octets describing a web page, it has to be copied in its entirety (and perhaps even transmitted) to the receiving program (if not a local program).

By contrast, for the call method, only an address of say 4 or 8 bytes needs to be passed for each argument and may even be passed in a general purpose register requiring zero additional storage and zero "transfer time". This of course is not possible for distributed systems since an (absolute) address - in the callers address space - is normally meaningless to the remote program (however, a relative address might in fact be usable if the callee had an *exact* copy of, at least some of, the callers memory in advance).

Examples of message passing style

- Actor model implementation
- Amorphous computing
- Antiobjects
- Flow-based programming
- SOAP (protocol)
- Smalltalk

Influences on other programming models

In the terminology of some object-oriented programming languages, a message is the single means to pass control to an object. If the object "responds" to the message, it has a method for that message. In pure object-oriented programming, message passing is performed exclusively through a dynamic dispatch strategy.

Objects can send messages to other objects from within their method bodies. Message passing enables extreme late binding in systems. Sending the same message to an object

twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical. Some languages support the forwarding or delegation of method invocations from one object to another if the former has no method to handle the message, but "knows" another object that may have one.

Alan Kay has argued that message passing is more important than objects in OOP, and that objects themselves are often over-emphasized. The live distributed objects programming model builds upon this observation; it uses the concept of a distributed data flow to characterize the behavior of a complex distributed system in terms of message patterns, using high-level, functional-style specifications.

Chapter 5

Abstraction and Encapsulation (Object-Oriented Programming)

Abstraction (computer science)

In computer science, the mechanism and practice of **abstraction** reduces and factors out details so that one can focus on a few concepts at a time.

The following English definition of abstraction helps to understand how this term applies to computer science, IT and objects:

abstraction - a concept or idea not associated with any specific instance

The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions; this has the fortunate effect of finessing some of the vexing philosophical issues of abstraction. For example, in both computing and in mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept.

In computer programming, abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subprograms and related concepts control flows
- Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype.

One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The

requirement that a programming language provide suitable abstractions is also called the abstraction principle.

Rationale

Computing mostly operates independently of the concrete world: The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentration on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's Tenth Rule is an aphorism on how such an architecture is both inevitable and complex.

A central form of abstraction in computing is language abstraction: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer languages* can be processed with a computer. An example of this abstraction process is the generational development of programming languages from the machine language to the assembly language and the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues for example in scripting languages and domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include the subroutine, the module, and the software component. Some other abstractions such as software design patterns and architectural styles remain invisible to a programming language and operate only in the design of a system.

Some abstractions try to limit the breadth of concepts a programmer needs by completely hiding the abstractions they in turn are built on. Joel Spolsky has criticised these efforts by claiming that all abstractions are *leaky* — that they can never completely hide the details below; however this does not negate the usefulness of abstraction. Some abstractions are designed to interoperate with others, for example a programming language may contain a foreign function interface for making calls to the lower-level language.

Language features

Programming languages

Different programming languages provide different types of abstraction, depending on the intended applications for the language. For example:

- In object-oriented programming languages such as C++, Object Pascal, or Java, the concept of **abstraction** has itself become a declarative statement - using the keywords `virtual` (in C++) or `abstract` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.

- Functional programming languages commonly exhibit abstractions related to functions, such as lambda abstractions (making a term into a function of some variable), higher-order functions (parameters are functions), bracket abstraction (making a term into a function of a variable).

Specification methods

Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages

Specification languages generally rely on abstractions of one kind or another, since specifications are typically defined earlier in a project (and at a more abstract level) than an eventual implementation. The UML specification language, for example, allows the definition of *abstract* classes, which remain abstract during the architecture and specification phase of the project.

Control abstraction

Programming languages offer control abstraction as one of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from one location of the memory to another location and producing the sum of two sequences of bits. Programming languages allow this to be done in the higher level. For example, consider this statement written in a Pascal-like fashion:

```
a := (1 + 2) * 5
```

To a human, this seems a fairly simple and obvious calculation (*"one plus two is three, times five is fifteen"*). However, the low-level steps necessary to carry out this evaluation, and return the value "15", and then assign that value to the variable "a", are actually quite subtle and complex. The values need to be converted to binary representation (often a much more complicated task than one would think) and the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of one register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of "15" to the variable labeled "a", so that "a" can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label

and the resultant location in physical or virtual memory, storing the binary representation of "15" to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time he simply wanted to add or multiply a couple of numbers and assign the result to a variable. Such duplication of effort has two serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed
2. it forces the programmer to program for the particular hardware and instruction set

Structured programming

Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control and interfaces between components, with reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points and (where possible) to have single exit points from functions and procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships and at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on and off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen and the database interface (which may itself be a standalone third party library or a statically linked set of library routines).
- Either the database or the payroll application also has to initiate the process of exchanging data with between ship and shore, and that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of one component and its assorted internal methods from the others. Object-oriented programming embraced and extended this concept.

Data abstraction

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—

while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. As one way to look at this: the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include Ada and Modula-2. Object-oriented languages are commonly claimed to offer data abstraction; however, their inheritance concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the Fragile binary interface problem.

Abstraction in object oriented programming

In object-oriented programming theory, **abstraction** involves the facility to define objects that represent abstract "actors" that can perform work, report on and change their state, and "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, and standardizing the way that different data types interact, is the beginning of **abstraction**. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various object-oriented programming languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of one type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Common Lisp Object System or self, for example, feature less of a class-instance distinction and more use of delegation for polymorphism. Individual objects and functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates and overloading and other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter the need to support abstract nouns in code - all programming relies on an ability to abstract verbs as functions, nouns as data structures, and either as processes.

Consider for example a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger and feeding. It defines an `Animal` class to represent both the state of the animal and its functions:

```
public class Animal extends LivingThing
{
     private Location loc;
     private double energyReserves;

     boolean isHungry() {
         return energyReserves < 2.5;
     }
     void eat(Food f) {
         // Consume food
         energyReserves += f.getCalories();
     }
     void moveTo(Location l) {
         // Move to new location
         loc = l;
     }
}
```

With the above definition, one could create objects of type `Animal` and call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class *`Animal`* is an abstraction used in place of an actual animal, *`LivingThing`* is a further abstraction (in this case a generalisation) of *`Animal`*.

If one requires a more differentiated hierarchy of animals — to differentiate, say, those who provide milk from those who provide nothing except meat at the end of their lives — that is an intermediary level of abstraction, probably DairyAnimal (cows, goats) who would eat foods suitable to giving good milk, and Animal (pigs, steers) who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so s/he could concentrate instead on the feeding schedule. The two classes could be related using inheritance or stand alone, and the programmer could define varying degrees of polymorphism between the two types. These facilities tend to vary drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design

Decisions regarding what to abstract and what to keep under the control of the coder become the major concern of object-oriented design and domain analysis—actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time and budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs and cows and their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available and thus there is no reason to code the type of food into the class itself, and the design is a single simple Animal class of which pigs and cows are instances with the same functions. A decision to differentiate DairyAnimal would change the detailed analysis but the domain and legacy analysis would be unchanged—thus it is entirely under the control of the programmer, and we refer to abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, **abstraction** refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. For instance, one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a **concrete** (more precise) model of execution.

Abstraction may be **exact** or **faithful** with respect to a property if one can answer a question about the property equally well on the concrete or abstract model. For instance, if we wish to know what the result of the evaluation of a mathematical expression involving only integers +, -, ×, is worth modulo n, we need only perform all operations modulo n (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily **exact**, should be **sound**. That is, it should be possible to get sound answers from them—even though the abstraction may simply yield a result of undecidability. For instance, we may abstract the students in a class by

their minimal and maximal ages; if one asks whether a certain person belongs to that class, one may simply compare that person's age with the minimal and maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction and other characteristics of the design, and how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because non-trivial properties of computer programs are essentially undecidable. As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

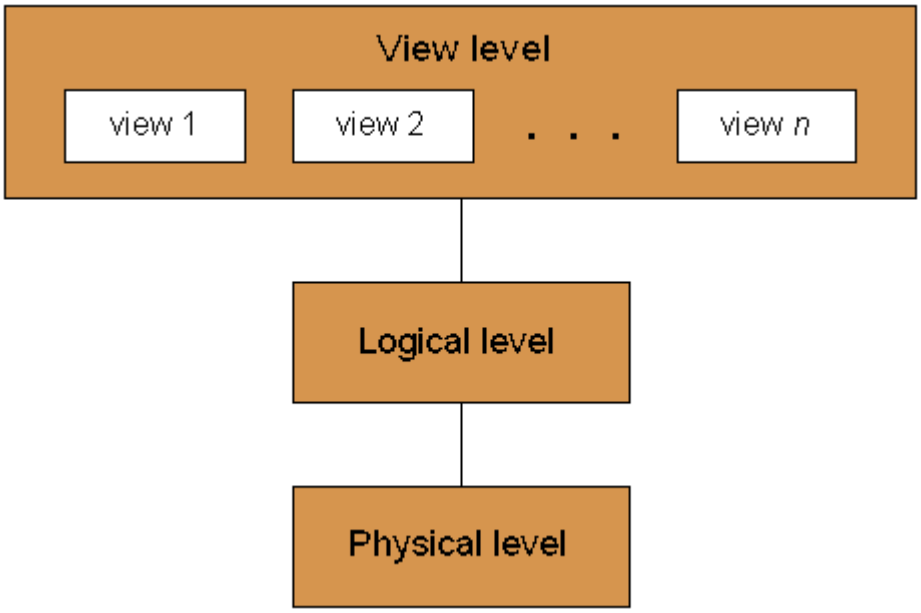
Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

Levels of abstraction

Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information and processes, but uses a system of expression involving a unique set of objects and compositions that apply only to a particular domain. Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. For example, gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications and operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems

Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels:



Data abstraction levels of a database system

Physical level: The lowest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.

Logical level: The next higher level of abstraction describes *what* data the database stores, and what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.

View level: The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Layered architecture

The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction

Systems design and business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). it is a technique used in designing computer software, hardware, and communications in which system or network components are isolated in layers so that changes can be made in one layer without affecting the others.

Encapsulation (object-oriented programming)

In an object-oriented programming language **encapsulation** is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting access to some of the object's components.
- A language construct that facilitates the bundling of data with the methods operating on that data.

Programming language researchers and academics generally use the first meaning alone or in combination with the second as a distinguishing feature of object oriented programming. The second definition is motivated by the fact that in many OOP languages hiding of components is not automatic or can be overridden; thus information hiding is defined as a separate notion by those who prefer the second definition.

As information hiding mechanism

Under this definition, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields. Some languages like Smalltalk and Ruby only allow access via object methods, but most others (e.g. C++ or Java) offer the programmer a degree of control over what is hidden, typically via keywords like `public` and `private`. It should be noted that the ISO C++ standard refers to `private` and `public` as "access specifiers" and that they do not "hide any information". Information hiding is accomplished by furnishing a compiled version of the source code that is interfaced via a header file.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.

Almost always there is a way to override such protection - usually via reflection API (Ruby, Java, C# etc.), sometimes by mechanism like name mangling (Python), or special

keyword like `friend` in C++. This access is often necessary for debuggers, serialization, unit testing, but not uncommonly in normal code when `private` was overeagerly applied.

This mechanism is not unique to object-oriented programming. Implementations of abstract data types, e.g. modules offer a similar form of encapsulation. This similarity stems from the fact that both notions rely on the same mathematical fundament of an existential type.

Chapter 6

Inheritance (Object-Oriented Programming) and Mutator Method

Inheritance (object-oriented programming)

In object-oriented programming (OOP), **inheritance** is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called objects which can be based on previously created objects. In *classical inheritance* where objects are defined by classes, classes can inherit other classes. The new classes, known as *sub-classes* (or derived classes), inherit attributes and behavior of the pre-existing classes, which are referred to as *super-classes* (or ancestor classes). The inheritance relationship of sub- and superclasses gives rise to a hierarchy. In Prototype-based programming objects can be defined directly from other objects without the need to define any classes.

The inheritance concept was invented in 1967 for Simula.

Inheritance should not be confused with (subtype) polymorphism, commonly called just polymorphism. Inheritance is a relationship between implementations, whereas subtype polymorphism is relationship between types (interfaces in OOP). (Compare connotation/denotation.) In some, but not all OOP languages, the notions coincide because the only way to declare a subtype is to define a new class that inherits the implementation of another.

Inheritance does not entail behavioral subtyping either. It is entirely possible to derive a class whose object will behave incorrectly when used in a context where the parent class is expected.

Complex inheritance, or inheritance used within an insufficiently mature design, may lead to the yo-yo problem.

Applications of inheritance

Inheritance is used to co-relate two or more classes to each other. With the use of inheritance we can use the methods and the instance variables of other classes in any other classes.

Overriding

Many object-oriented programming languages permit a class or object to replace the implementation of an aspect—typically a behavior—that it has inherited. This process is usually called *overriding*. Overriding introduces a complication: which version of the behavior does an instance of the inherited class use—the one that is part of its own class, or the one from the parent (base) class? The answer varies between programming languages, and some languages provide the ability to indicate that a particular behavior is not to be overridden and behave according to the base class.

Code reuse

One of the earliest motivations for using inheritance was the reuse of code which already existed in another class. This practice is usually called implementation inheritance.

In most quarters, class inheritance for the sole purpose of code reuse has fallen out of favor. The primary concern is that implementation inheritance does not provide any assurance of polymorphic substitutability—an instance of the reusing class cannot necessarily be substituted for an instance of the inherited class. An alternative technique, delegation, requires more programming effort but avoids the substitutability issue. In C++ private inheritance can be used as form of *implementation inheritance* without substitutability. Whereas public inheritance represents an "is-a" relationship and delegation represents a "has-a" relationship, private (and protected) inheritance can be thought of as an "is implemented in terms of" relationship.

Object Oriented-Software Construction, 2nd edition by Bertrand Meyer, the creator of the object-oriented programming language Eiffel, lists twelve different uses of inheritance , most of which involve some amount of implementation inheritance.

Inheritance vs subtype polymorphism

There are many kinds of polymorphisms (in the type system of a language). For example,

- templates (generics),
- member/operator overloading,
- coercion, and
- subtype polymorphism.

When someone says polymorphism in association with object orientation it's often subtype polymorphism that's meant. In principle subtype polymorphism means that an object can be of many types. Like

```
class B : public A {};
```

B objects arent's just B objects, they're A objects too. This is achieved by the way of inheritance.

Now you can also do,

```
class C : public A {};
```

and thanks to subtype polymorphism C objects are also A objects.

The situation now is that because both C and B objects are also A objects they can both be assigned to A type variables. This is what subtype polymorphism accomplishes. But this would be quite meaningless if the B and C objects were identical. Fortunately they don't have to be, because thanks to overriding they can be made to behave differently.

So the principal language mechanism behind subtype polymorphism is inheritance (one object can have many types). The role of overriding is to allow subtype objects sharing a common supertype to behave differently.

Limitations and alternatives

When using inheritance extensively in designing a program, one should note certain constraints that it imposes.

For example, consider a class `Person` that contains a person's name, address, phone number, age, gender, and race. We can define a subclass of `Person` called `Student` that contains the person's grade point average and classes taken, and another subclass of `Person` called `Employee` that contains the person's job-title, employer, and salary.

In defining this inheritance hierarchy we have already defined certain restrictions, not all of which are desirable:

Constraints of inheritance-based design

- **Singleness**: using single inheritance, a subclass can inherit from only one superclass. Continuing the example given above, `Person` can be either a `Student` or an `Employee`, but not both. Using multiple inheritance partially solves this problem, as one can then define a `StudentEmployee` class that inherits from both `Student` and `Employee`. However, it can still inherit from each superclass only once; this scheme does not support cases in which a student has two jobs or attends two institutions.

- **Static**: the inheritance hierarchy of an object is fixed at instantiation when the object's type is selected and does not change with time. For example, the inheritance graph does not allow a `Student` object to become a `Employee` object while retaining the state of its `Person` superclass. (Although similar behavior can be achieved with the decorator pattern.) Some have criticized inheritance, contending that it locks developers into their original design standards.

- **Visibility**: whenever client code has access to an object, it generally has access to all the object's superclass data. Even if the superclass has not been declared public, the client can still cast the object to its superclass type. For example, there is no way to give a function a pointer to a `Student`'s grade point average and transcript without also giving that function access to all of the personal data stored in the student's `Person` superclass.

The Composite reuse principle is an alternative to inheritance. This technique supports polymorphism and code reuse by separating behaviors from the primary class hierarchy and including specific behavior classes as required in any business domain class. This approach avoids the static nature of a class hierarchy by allowing behavior modifications at run time and allows a single class to implement behaviors buffet-style, instead of being restricted to the behaviors of its ancestor classes.

Roles and inheritance

Sometimes inheritance-based design is used instead of roles. A role, say **Student** role of a **Person** describes a characteristic associated to the object that is present because the object happens to participate in some relationship with another object (say the person in student role -has enrolled- to the classes). Some object-oriented design methods do not distinguish this use of roles from more stable aspects of objects. Thus there is a tendency to use inheritance to model roles, say you would have a Student role of a Person modelled as a subclass of a Person. However, neither the inheritance hierarchy nor the types of the objects can change with time. Therefore, modelling roles as subclasses can cause the roles to be fixed on creation, say a Person cannot then easily change his role from Student to Employee when the circumstances change. From modelling point of view, such restrictions are often not desirable, because this causes artificial restrictions on future extensibility of the object system, which will make future changes harder to implement, because existing design needs to be updated. Inheritance is often better used with a generalization mindset, such that common aspects of instantiable classes are factored to superclasses; say having a common superclass 'LegalEntity' for both Person and Company classes for all the common aspects of both. The distinction between role based design and inheritance based design can be made based on the stability of the aspect. Role based design should be used when it's conceivable that the same object participates in different roles at different times, and inheritance based design should be used when the common aspects of multiple classes (not objects!) are factored as superclasses, and do not change with time.

One consequence of separation of roles and superclasses is that this cleanly separates compile-time and run-time aspects of the object system. Inheritance is then clearly a compile-time construct. Inheritance does influence the structure of many objects at run-time, but the different kinds of structure that can be used are already fixed at compile-time.

To model the example of `Person` as an employee with this method, the modelling ensures that a `Person` class can only contain operations or data that are common to every Person

instance regardless of where they are used. This would prevent use of a Job member in a Person class, because every person does not have a job, or at least it is not known that the Person class is only used to model Person instances that have a job. Instead, object-oriented design would consider some subset of all person objects to be in an "employee" role. The job information would be associated only to objects that have the employee role. Object-oriented design would also model the "job" as a role, since a job can be restricted in time, and therefore is not a stable basis for modelling a class. The corresponding stable concept is either "WorkPlace" or just "Work" depending on which concept is meant. Thus, from object-oriented design point of view, there would be a "Person" class and a "WorkPlace" class, which are related by a many-to-many associatation "works-in", such that an instance of a Person is in employee role, when he works-in a job, where a job is a role of his work place in the situation when the employee works in it.

Note that in this approach, all classes that are produced by this design process form part of the same domain, that is, they describe things clearly using just one terminology. This is often not true for other approaches.

The difference between roles and classes is especially difficult to understand if one assumes referential transparency, because roles are types of references and classes are types of the referred-to objects.

Mutator method

In computer science, a **mutator method** is a method used to control changes to a variable.

The mutator method, sometimes called a "setter", is most often used in object-oriented programming, in keeping with the principle of encapsulation. According to this principle, member variables of a class are made private to hide and protect them from other code, and can only be modified by a public member function (the mutator method), which takes the desired new value as a parameter, optionally validates it, and modifies the private member variable.

Often a "setter" is accompanied by a "getter" (also known as an **accessor**), which returns the value of the private member variable.

Mutator methods may also be used in non-object-oriented environments. In this case, a reference to the variable to be modified is passed to the mutator, along with the new value. In this scenario, the compiler cannot restrict code from bypassing the mutator method and changing the variable directly. The onus falls to the developers to ensure the variable is only modified through the mutator method and not modified directly.

In programming languages that support them, properties offer a convenient alternative without giving up the utility of encapsulation.

In the examples below, a fully implemented mutator method can also validate the input data or take further action such as triggering an event.

Implications

The alternative to defining mutator and accessor methods, or property blocks, is to give the instance variable some visibility other than private and access it directly from outside the objects. Much finer control of access rights can be defined using mutators and accessors. For example, a parameter may be made read-only simply by defining an accessor but not a mutator. The visibility of the two methods may be different, it is often useful for the accessor is public while the mutator remains protected, package-private or internal.

The block where the mutator is defined provides an opportunity for validation or preprocessing of incoming data. If all external access is guaranteed to come through the mutator, then these steps cannot be bypassed. For example, if a date is represented by separate private `year`, `month` and `day` variables, then incoming dates can be split by the `setDate` mutator while for consistency the same private instance variables are accessed by `setYear` and `setMonth`. In all cases month values outside of 1 - 12 can be rejected by the same code.

Accessors conversely allow for synthesis of useful data representations from internal variables while keeping their structure encapsulated and hidden from outside modules. A monetary `getAmount` accessor may build a string from a numeric variable with the number of decimal places defined by a hidden `currency` parameter.

Modern programming languages often offer the ability to generate the boilerplate for mutators and accessors in a single line—as for example C#'s `public string Name { get; set; }` and Ruby's `attr_accessor :name`. In these cases, no code blocks are created for validation, preprocessing or synthesis. These simplified accessors still retain the advantage of encapsulation over simple public instance variables, but it is common that, as system designs progress, the software is maintained and requirements change, the demands on the data become more sophisticated. Many automatic mutators and accessors eventually get replaced by separate blocks of code. The benefit of automatically creating them in the early days of the implementation is that the public interface of the class remains identical whether or not greater sophistication is added, requiring no extensive refactoring if it is.

Manipulation of parameters that have mutators and accessors from *inside* the class where they are defined often requires some additional thought. In the early days of an implementation, when there is little or no additional code in these blocks, it makes no difference if the private instance variable is accessed directly or not. As validation, cross-validation, data integrity checks, preprocessing or other sophistication is added, subtle

bugs may appear where some internal access makes use of the newer code while in other places it is bypassed.

Accessor functions are always less efficient than directly fetching or storing data fields due to the extra steps involved.

Smalltalk example

```
  age: aNumber
     " Set the receiver age to be aNumber if is greater than 0 and less
than 150 "
    (aNumber between: 0 and: 150)
       ifTrue: [ age := aNumber ]
```

C example

Note that it is perfectly possible to do object-oriented programming with guaranteed encapsulation in pure C.

In file student.h:

```
typedef struct student *student_t;
student_t student_new( int age, char *name);
void student_set_age( student_t s, int age);
int student_get_age( student_t s);
```

In file student.c:

```
#include <student.h>
struct student { int age; char * name; };

student_t student_new( int age, char *name) {
  student_t s = malloc( sizeof( struct student));
  s->age = age; s->name = name;
  return s;
}
void student_set_age( student_t s, int age) { s->age = age; }
int student_get_age( student_t s) { return s->age; }
```

In file main.c:

```
#include <student.h>

int main() {
 student_t s = student_new(19, "Maurice");
 int old_age = student_get_age(s);
 student_set_age(s, 21);
}
```

C++ example

In file Point.h

```
#ifndef POINT_H
#define POINT_H

class Point {
private:
    int _x;
    int _y;
public:

    //Constructors
    //...

    //Getters (Accessors)
    int GetX() const;
    int GetY() const;

    //Setters (Mutators)
    void SetX(int x);
    void SetY(int y);

    //Methods
    //...
};
#endif
```

In file Point.cpp
This example optionally validates the setters.

```
#include "Point.h"

#include <climits>

//Constructors
//...

//Getters (Accessors)
int Point::GetX() const{
    return _x;
}
int Point::GetY() const{
    return _y;
}
//Setters (Mutators}
void Point::SetX(int x) {
    if(x < 0) x = 0;
    _x = x;
}
void Point::SetY(int y) {
    if(y < 0) y = 0;
    _y = y;
}
//Methods
//...
```

Python example

This example uses a Python class with one variable, a getter, and a setter.

```
class Student():
    # A variable to hold the students name
    _name = ""

    # A function to print the name to stdout
    def getName(self):
        print self._name

    # A function to set a new name
    def setName(self, newName):
        self._name = newName
```

Java example

In this example of a simple class representing a student with only the name stored, one can see the variable *name* is private, i.e. only visible from the Student class, and the "setter" and "getter" is public, namely the "getName()" and "setName(name)" methods.

```
public class Student {
    private String name;

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param newName
     * the name to set
     */
    public void setName(String newName) {
        name = newName;
    }
}
```

C# example

This example illustrates the C# idea of properties, which are a special type of class member. Unlike Java, no explicit methods are defined; a public 'property' contains the logic to handle the actions. Note use of the built-in (undeclared) variable `value`.

```
public class Student
{
    private string name;

    /// <summary>
```

```
    /// gets or sets student's name
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}
```

In recent C# versions, this example may be abbreviated as follows, without declaring the private variable `name`.

```
public class Student
{
    public string Name { get; set; }
}
```

Using the abbreviated syntax means that the underlying variable is no longer available from inside the class. As a result, the `set` portion of the property must be present for assignment. Access can be restricted with a `set`-specific access modifier.

```
public class Student
{
    public string Name { get; private set; }
}
```

VB.NET example

This example illustrates the VB.NET idea of properties, which are used in classes. Similar to C#, there is an explicit use of the `Get` and `Set` methods.

```
Public Class Student

    Private _name As String

    Public Property Name()
        Get
            Return _name
        End Get
        Set(ByVal value)
            _name = value
        End Set
    End Property

End Class
```

In VB.NET 2010, Auto Implemented properties can be utilized to create a property without having to use the Get and Set syntax. Note that a hidden variable is created by the compiler, called `_name`, to correspond with the Property `name`. Using another variable within the class named `_name` would result in an error. Privileged access to the underlying variable is available from within the class.

```
Public Class Student
    Public Property name As String
End Class
```

Delphi example

This is a simple class in Delphi language that illustrates the concept of public property that access a private field.

```
interface

type
    TStudent = class
        private
            fName: string;
            procedure SetName(const Value: String);
        public
            /// <summary>
            /// Set or get the name of the student.
            /// </summary>
            property Name:String read fName write fName;
    end;
```

PHP example

It this example of a simple class representing a student with only the name stored, one can see the variable *name* is private, i.e. only visible from the Student class, and the "setter" and "getter" is public, namely the "getName()" and "setName('name')" methods.

```
class Student {
    private $name;

    /**
     * @return the $name
     */
    public function getName() {
        return $this->name;
    }

    /**
     * @param $newName
     * the name to set
     */
    public function setName($newName) {
        $this->name = $newName;
    }
```

```
}
```

Perl example

```
package Student;

sub new {
        bless {}, shift;
}

sub set_name {
        my $self = shift;
        $self->{name} = $_[0];
}

sub get_name {
        my $self = shift;
        return $self->{name};
}

1;
```

Or, using Class::Accessor

```
package Student;
use base qw(Class::Accessor);
__PACKAGE__->follow_best_practice;

Student->mk_accessors(qw(name));

1;
```

Or, using Moose Object System:

```
package Student;
use Moose;

# Moose uses the attribute name as the setter and getter, the reader
and writer properties
# allow us to override that and provide our own names, in this case
get_name and set_name
has 'name' => (is => 'rw', isa => 'Str', reader => 'get_name', writer
=> 'set_name');

1;
```

Ruby example

In Ruby, individual accessor and mutator methods may be defined, or the metaprogramming constructs `attr_reader` or `attr_accessor` may be used both to declare a private variable in a class and to provide either read-only or read-write public access to it respectively.

Defining individual accessor and mutator methods creates space for pre-processing or validation of the data

```
class Student
  def name
    @name
  end

  def name=(value)
    @name=value
  end
end
```

Read-only simple public access to implied `@name` variable

```
class Student
  attr_reader :name
end
```

Read-write simple public access to implied `@name` variable

```
class Student
  attr_accessor :name
end
```

Chapter 7

Constructor (Object-Oriented Programming) and Dependency Injection

Constructor (object-oriented programming)

In object-oriented programming, a **constructor** (sometimes shortened to **ctor**) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created.

A constructor resembles an instance method, but it differs from a method in that it never has an explicit return-type, it is not inherited (though many languages provide access to the superclass's constructor,for example through the `super` keyword in Java), and it usually has different rules for scope modifiers. Constructors are often distinguished by having the same name as the declaring class. They have the task of initializing the object's data members and of establishing the invariant of the class, failing if the invariant isn't valid. A properly written constructor will leave the object in a *valid* state. Immutable objects must be initialized in a constructor.

Programmers can also use the term *constructor* to denote one of the tags that wraps data in an algebraic data type.

Most languages allow overloading the constructor in that there can be more than one constructor for a class, each having different parameters. Some languages take consideration of some special types of constructors:

- default constructor – a constructor that can take no arguments
- copy constructor – a constructor that takes one argument of the type of the class (or a reference thereof)

Java

In Java, some of the differences between other methods and constructors are:

- Constructors never have an explicit return type.

- Constructors cannot be directly invoked (the keyword "`new`" must be used).
- Constructors cannot be synchronized, final, abstract, native, or static.
- Constructors are always executed by the same thread.

Apart from this, a Java constructor performs the following functions in the following order:

1. It initializes the class variables to default values. (Byte, short, int, long, float, and double variables default to their respective zero values, booleans to false, chars to the null character ('\u0000') and references of any objects to null.)
2. It then calls the super class constructor (default constructor of super class only if no constructor is defined).
3. It then initializes the class variables to the specified values like ex: int var = 10; or float var = 10.0f and so on.
4. It then executes the body of the constructor.

Example

```
public class Example
{
  //definition of the constructor.
  public Example()
  {
    this(1);
  }

  //overloading a constructor
  public Example(int input)
  {
    data = input; //This is an assignment
  }

  //declaration of instance variable(s).
  private int data;
}
//code somewhere else
//instantiating an object with the above constructor
Example e = new Example(42);
```

Visual Basic .NET

In Visual Basic .NET, constructors use a method declaration with the name "`New`".

Example

```
Class Foobar
  Private strData As String

  ' Constructor
  Public Sub New(ByVal someParam As String)
     strData = someParam
```

```
  End Sub
End Class
' code somewhere else
' instantiating an object with the above constructor
Dim foo As New Foobar(".NET")
```

C#

In C#, a constructor is thus.

Example

```
public class MyClass
{
  private int a;
  private string b;

  //constructor
  public MyClass() : this(42, "string")
  {
  }

  //overloading a constructor
  public MyClass(int a, string b)
  {
    this.a = a;
    this.b = b;
  }
}
//code somewhere
//instantiating an object with the constructor above
MyClass c = new MyClass(42, "string");
```

C# static constructor

In C#, a static constructor is a static data initializer. Static constructors allow complex static variable initialization. Static constructors can be called once and call is made implicitly by the run-time right before the first time the class is accessed. Any call to a class (static or constructor call), triggers the static constructor execution. Static constructors are thread safe and are a great way to implement a singleton pattern. When used in a generic programming class, static constructors are called on every new generic instantiation one per type (static variables are instantiated as well).

Example

```
public class MyClass
{
  private static int _A;

  //normal constructor
  static MyClass()
  {
```

```
    _A = 32;
  }

  //standard default constructor
  public MyClass()
  {

  }
}
//code somewhere
//instantiating an object with the constructor above
//right before the instantiation
//the variable static constructor is executed and _A is 32
MyClass c = new MyClass();
```

C++

In C++, the name of the constructor is the name of the class. It can have parameters like any member functions (methods).

The constructor has two parts. First is the initializer list which comes after the parameter list and before the opening curly bracket of the method's body. It starts with a colon and separated by commas. You are not always required to have initializer list, but it gives the opportunity to construct data members with parameters so you can save time (one construction instead of a construction and an assignment). Sometimes you must have initializer list for example if you have *const* or reference type data members, or members that are cannot be default constructed (they don't have parameterless constructor). The order of the list should be the order of the declaration of the data members, because the execution order is that. The second part is the body which is a normal method body surrounded by curly brackets.

C++ allows more than one constructor. The other constructors cannot be called, but can have default values for the parameters. The constructor of a base class (or base classes) can also be called at the start of the initializer list. Virtual functions cannot be called from the constructor.

A copy constructor has a parameter of the same type passed as *const* reference, for example *Vector(const Vector& rhs)*. If it is not implemented by hand the compiler gives a default implementation which uses the copy constructor for each member variable or simply copies values in case of primitive types. The default implementation is not efficient if the class has dynamically allocated members (or handles to other resources), because it can lead to double calls to *delete* (or double release of resources) upon destruction.

Example

```
class Vector {
public:
```

```
    Vector(double r = 1.0, double alpha = 0.0) // Constructor,
parameters with default values.
    : x(r*cos(alpha)) // <- Initializer list
    {
        y = r*sin(alpha); // <- Normal assignment
    }
    // Other member functions
private:
    double x; // Data members, they should be private
    double y;
};
```

Example invocations:

```
Example a,b(3),c(5,M_PI/4);
```

F#

In F#, a constructor can include any `let` or `do` statements defined in a class. `let` statements define private fields and `do` statements execute code. Additional constructors can be defined using the `new` keyword.

Example

```
type MyClass(_a : int, _b : string) = class
    // primary constructor
    let a = _a
    let b = _b
    do printfn "a = %i, b = %s" a b

    // additional constructors
    new(_a : int) = MyClass(_a, "") then
        printfn "Integer parameter given"

    new(_b : string) = MyClass(0, _b) then
        printfn "String parameter given"

    new() = MyClass(0, "") then
        printfn "No parameter given"
end
//code somewhere
//instantiating an object with the primary constructor
let c1 = new MyClass(42, "string")

//instantiating an object with additional constructors
let c2 = new MyClass(42)
let c3 = new MyClass("string")
let c4 = MyClass() // "new" keyword is optional
```

Eiffel

In Eiffel, the routines which initialize new objects are called **creation procedures**. They are similar to constructors in some ways and different in others. Creation procedures have the following traits:

- Creation procedures never have an explicit return type (by definition of **procedure**).
- Creation procedures are named. Names are restricted only to valid identifiers.
- Creation procedures are designated by name as creation procedures in the text of the class.
- Creation procedures can be directly invoked to re-initialize existing objects.
- Every effective (i.e., concrete or non-abstract) class must designate at least one creation procedure.
- Creation procedures must leave the newly initialized object in a state that satisfies the class invariant.

Although object creation involves some subtleties, the creation of an attribute with a typical declaration `x: T` as expressed in a creation instruction `create x.make` consists of the following sequence of steps:

- Create a new direct instance of type `T`.
- Execute the creation procedure `make` to the newly created instance.
- Attach the newly initialized object to the entity `x`.

Example

In the first snippet below, class `POINT` is defined. The procedure `make` is coded after the keyword `feature`.

The keyword `create` introduces a list of procedures which can be used to initialize instances. In this case the list includes `default_create`, a procedure with an empty implementation inherited from class `ANY`, and the `make` procedure coded within the class.

```
class
    POINT
create
    default_create, make

feature

    make (a_x_value: REAL; a_y_value: REAL)
        do
            x := a_x_value
            y := a_y_value
        end

    x: REAL
```

```
            -- X coordinate

    y: REAL
            -- Y coordinate
        ...
```

In the second snippet, a class which is a client to `POINT` has a declarations `my_point_1` and `my_point_2` of type `POINT`.

In procedural code, `my_point_1` is created as the origin (0.0, 0.0). Because no creation procedure is specified, the procedure `default_create` inherited from class `ANY` is used. This line could have been coded `create my_point_1.default_create` . Only procedures named as creation procedures can be used in an instruction with the `create` keyword. Next is a creation instruction for `my_point_2`, providing initial values for the `my_point_2`'s coordinates. The third instruction makes an ordinary instance call to the `make` procedure to reinitialize the instance attached to `my_point_2` with different values.

```
    my_point_1: POINT
    my_point_2: POINT
        ...

            create my_point_1
            create my_point_2.make (3.0, 4.0)
            my_point_2.make (5.0, 8.0)
        ...
```

ColdFusion

ColdFusion has no constructor method. Developers using it commonly create an '`init`' method that acts as a pseudo-constructor.

Example

```
<cfcomponent displayname="Cheese">
   <!--- properties --->
   <cfset variables.cheeseName = "" />
   <!--- pseudo-constructor --->
   <cffunction name="init" returntype="Cheese">
      <cfargument name="cheeseName" type="string" required="true" />
      <cfset variables.cheeseName = arguments.cheeseName />
      <cfreturn this />
   </cffunction>
</cfcomponent>
```

Pascal

In Object Pascal, the constructor is a method named `create`, which the keyword `constructor` automatically calls after creating the object. It is usually used to automatically perform various initializations such as property initializations. Constructors

can also accept arguments, in which case, when the `create` statement is written, you also need to send the constructor the function parameters in between the parentheses.

Example

```
type
  TPerson = class
  private
    FName: String;
  public
    property Name: String read FName;
    constructor Create(AName: String);
  end;

implementation

constructor TPerson.Create(AName: String);
begin
  FName := AName;
end;
```

Perl

In Perl version 5, by default, constructors must provide code to create the object (a reference, usually a hash reference, but sometimes an array reference, scalar reference or code reference) and bless it into the correct class. By convention the constructor is named *new*, but it is not required, or required to be the only one. For example, a Person class may have a constructor named *new* as well as a constructor *new_from_file* which reads a file for Person attributes, and *new_from_person* which uses another Person object as a template.

Example

```
package Person;
use strict;
use warnings;

# constructor
sub new {
        # class name is passed in as 0th
        # argument
        my $class = shift;
        # check if the arguments to the
        # constructor are key => value pairs
        die "$class needs arguments as key => value pairs"
                unless (@_ % 2 == 0);
        # default arguments
        my %defaults;

        # create object as combination of default
        # values and arguments passed
        my $obj = {
```

```
                %defaults,
                @_,
        };
        # check for required arguments
        die "Need first_name and last_name for Person"
                unless ($obj->{first_name} and $obj->{last_name});
         # any custom checks of data
         if ($obj->{age} && $obj->{age} < 18)) { # no under-18s
                 die "No under-18 Persons";
         }
        # return object blessed into Person class
        bless $obj, $class;
}
1;
```

Perl with Moose

With the Moose object system for Perl, most of this boilerplate can be left out, a default *new* is created, attributes can be specified, as well as whether they can be set, reset, or are required. In addition, any extra constructor functionality can be included in a *BUILD* method which the Moose generated constructor will call, after it has checked the arguments. A *BUILDARGS* method can be specified to handle constructor arguments not in hashref / key => value form.

Example

```
package Person;
# enable Moose-style object construction
use Moose;

# first name ( a string) can only be set at construction time ('ro')
has first_name => (is => 'ro', isa => 'Str', required => 1);
# last name ( a string) can only be set at construction time ('ro')
has last_name  => (is => 'ro', isa => 'Str', required => 1);
# age (Integer) can be modified after construction ('rw'), and is not
required
# to be passed to be constructor.  Also creates a 'has_age' method
which returns
# true if age has been set
has age        => (is => 'rw', isa => 'Int', predicate => 'has_age');

# Check custom requirements
sub BUILD {
      my $self = shift;
      if ($self->has_age && $self->age < 18) { # no under 18s
           die "No under-18 Persons";
      }
}
1;
```

In both cases the Person class is instiated like this:

```
use Person;
my $p = Person->new( first_name => 'Sam', last_name => 'Ashe', age =>
42 );
```

PHP

In PHP (version 5 and above), the constructor is a method named `__construct()`, which the keyword `new` automatically calls after creating the object. It is usually used to automatically perform various initializations such as property initializations. Constructors can also accept arguments, in which case, when the `new` statement is written, you also need to send the constructor the function parameters in between the parentheses.

Example

```
class person
{
   private $name;

   function __construct($name)
   {
       $this->name = $name;
   }

   function getName()
   {
       return $this->name;
   }
}
```

However, constructor in PHP version 4 (and earlier) is a method in a class with the same name of the class.

```
class Person
{
   private $name;

   function Person($name)
   {
       $this->name = $name;
   }

   function getName()
   {
       return $this->name;
   }
}
```

Python

In Python, constructors are created by defining an __init__ method, and are called when a new instance is created by calling the class. Unlike other languages such as C++, derived classes in Python do not call their base classes' constructors. However, when a

constructor is not defined, the next one found in the class's Method Resolution Order will be called. Due to Python's use of duck typing, class members are often defined in the constructor, rather than in the class definition itself.

Example

```
class ExampleClass:
    def __init__(self, number):
        self.number = number
        print "Here is a number: %d" % self.number

exampleInstance = ExampleClass(420)
```

Constructors simplified, with pseudocode

Constructors are always part of the implementation of classes. A class (in programming) refers to a specification of the general traits of the set of objects that are members of the class rather than the specific traits of any object at all. A simple analogy in pseudocode follows. Consider the set (or class, using its generic meaning) of students at some school. Thus we have

```
class Student {
    // refers to the class of students
    // ... more omitted ...
}
```

However, the class `Student` just provides a generic prototype of what a student should be. To use it, the programmer creates each student as an *object* or *instance* of the class. This object is a real quantity of data in memory whose size, layout, traits, and (to some extent) behavior are determined by the class definition. The usual way of creating objects is to call a constructor (classes may in general have many independent constructors). For example,

```
class Student {
    Student (String studentName, String Address, int ID) {
        // ... storage of input data and other internal fields here ...
    }
    // ...
}
```

Dependency injection

Dependency injection (**DI**) in object-oriented computer programming is a technique that indicates to a part of a program which other parts it can use, i.e. to supply an external dependency (i.e. a reference) to a software component. In technical terms, it is a design pattern that separates behavior from dependency resolution, thus decoupling highly dependent components.

Developers of software strive to reduce dependencies between components in software for various reasons. This leads to a new problem, though: How can a component know all the other components it needs to fulfill its purpose?

The traditional approach was to hard-code the dependency. As soon as the database driver was necessary, the component would execute a piece of code that would load a specific driver, configure it and call the necessary methods to interact with the database. If a second database must be supported, this piece of code would have to be modified or, even worse, copied and modified (violating the DRY principle).

Dependency injection offers a solution. Instead of hard-coding the dependencies, a component just lists the necessary services and a DI framework supplies these. At runtime, an independent component will load and configure the database driver and offer a standard interface to interact with the database. Again, the details have been moved from the original component to a set of new, small, database specific components, reducing the complexity of them all.

In DI terms, these new components are called "service components" because they render a service (database access) for one or more other components.

Dependency injection is a specific form of inversion of control where the concern being inverted is the process of obtaining the needed dependency. The term was first coined by Martin Fowler to describe the mechanism more clearly.

Basics

Without dependency injection, a consumer component that needs a particular service in order to accomplish a task will depend not only on the interface of the service but also on the details of a particular implementation of that service. The user component has to handle both its use of the service and the life-cycle of that service - creating an instance, opening and closing streams, disposing of unneeded objects, etc.

Using dependency injection, however, the life-cycle of a service is handled by a dependency provider rather than the consumer. The dependency provider is an independent, external component that links the consuming component and the providing component. The consumer would thus only need a reference to an implementation of the service that it needed in order to accomplish the necessary task.

Such a pattern involves at least three elements: a **dependent** consumer, the definition of its service **dependencies,** and an **injector** (sometimes referred to as a **provider** or **container**). The dependent is a consumer component that needs to accomplish a task in a computer program. In order to do so, it needs the help of various services (the dependencies) that execute certain sub-tasks. The provider is the component that is able to compose the dependent and its dependencies so that they are ready to be used, while also managing these objects' life-cycles. The provider may be implemented, for example, as a service locator, an abstract factory, a factory method or a more complex abstraction such as a framework.

The following is an example. A car (the consumer) depends upon an engine (the dependency) in order to move. The car's engine is made by an automaker (the dependency provider). The car does not know how to install an engine into itself, but it needs an engine in order to move. The automaker installs an engine into the car and the car utilizes the engine to move.

When the concept of dependency injection is used, it decouples high-level modules from low-level services. The result is called the dependency inversion principle.

Code illustration using Java

Using the car/engine example above mentioned, the following Java examples show how coupled dependencies (manually-injected dependencies) and framework-injected dependencies are typically staged.

```
public interface Engine {
    public float getEngineRPM();

    public void setFuelConsumptionRate(float flowInGallonsPerMinute);
}

public interface Car {
    public float getSpeedInMPH();

    public void setPedalPressure(float pedalPressureInPounds);
}
```

Highly coupled dependency

The following shows a common arrangement **with no dependency injection applied**:

```
public class DefaultEngineImpl implements Engine {
    private float engineRPM = 0;

    public float getEngineRPM() {
        return engineRPM;
    }

    public void setFuelConsumptionRate(float flowInGallonsPerMinute) {
        engineRPM = ...;
```

```
    }
}

public class DefaultCarImpl implements Car {
    private Engine engine = new DefaultEngineImpl();

    public float getSpeedInMPH() {
        return engine.getEngineRPM() * ...;
    }

    public void setPedalPressure(float pedalPressureInPounds) {
        engine.setFuelConsumptionRate(...);
    }
}

public class MyApplication {
    public static void main(String[] args) {
        Car car = new DefaultCarImpl();
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the above example, the `Car` class creates an instance of an `Engine` implementation in order to perform operations on the car. Hence, it is considered highly coupled because it couples a car directly with a particular engine implementation.

In cases where the `DefaultEngineImpl` dependency is managed outside of the scope of the `Car` class, the `Car` class must not instantiate the `DefaultEngineImpl` dependency. Instead, that dependency is injected externally.

Manually-injected dependency

Refactoring the above example to use manual injection:

```
public class DefaultCarImpl implements Car {
    private Engine engine;

    public DefaultCarImpl(Engine engineImpl) {
        engine = engineImpl;
    }

    public float getSpeedInMPH() {
        return engine.getEngineRPM() * ...;
    }

    public void setPedalPressure(float pedalPressureInPounds) {
        engine.setFuelConsumptionRate(...);
    }
}

public class CarFactory {
    public static Car buildCar() {
        return new DefaultCarImpl(new DefaultEngineImpl());
```

```
    }
}

public class MyApplication {
    public static void main(String[] args) {
        Car car = CarFactory.buildCar();
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the example above, the `CarFactory` class assembles a car and an engine together by injecting a particular engine implementation into a car. This moves the dependency management from the `Car` class into the `CarFactory` class. As a consequence, if the `Car` needed to be assembled with a different `Engine` implementation, the `Car` code would not be changed.

In a more realistic software application, this may happen if a new version of a base application is constructed with a different service implementation. Using factories, only the service code and the Factory code would need to be modified, but not the code of the multiple users of the service.

However, this still may not be enough abstraction for some applications, since in a realistic application there would be multiple Factory classes to create and update.

Framework-managed dependency injection

There are several frameworks available that automate dependency management by delegating the management of dependencies. Typically, this is accomplished by a Container using XML or "meta data" definitions. Refactoring the above example to use an external XML-definition framework:

```
    <service-point id="CarBuilderService">
        <invoke-factory>
            <construct class="Car">
                <service>DefaultCarImpl</service>
                <service>DefaultEngineImpl</service>
            </construct>
        </invoke-factory>
    </service-point>
/** Implementation not shown **/

public class MyApplication {
    public static void main(String[] args) {
        Service service =
(Service)DependencyManager.get("CarBuilderService");
        Car car = (Car)service.getService(Car.class);
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the above example, a dependency injection service is used to retrieve a `CarBuilderService` service. When a Car is requested, the service returns an appropriate implementation for both the car and its engine.

As there are many ways to implement dependency injection, only a small subset of examples is shown herein. Dependencies can be registered, bound, located, externally injected, etc., by many different means. Hence, moving dependency management from one module to another can be accomplished in a plethora of ways. However, there should exist a definite reason for moving a dependency away from the object that needs it because doing so can complicate the code hierarchy to such an extent that its usage appears to be "magical". For example, suppose a Web container is initialized with an association between two dependencies and that a user who wants to use one of those dependencies is unaware of the association. The user would thus not be able to detect any linkage between those dependencies and hence might cause drastic problems by using one of those dependencies.

Benefits and drawbacks

One benefit of using the dependency injection approach is the reduction of boilerplate code in the application objects since all work to initialize or setup dependencies is handled by a provider component.

Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in unit testing because it is easy to inject a fake implementation of a service into the object being tested by changing the configuration file.

One drawback is that excessive or inappropriate use of dependency injection can make applications more complicated, harder to understand and more difficult to modify. Code that uses dependency injection can seem *magical* to some developers, since instantiation and initialization of objects is handled completely separately from the code that uses it. This separation can also result in problems that are hard to diagnose. Additionally, some dependency injection frameworks maintain verbose configuration files, requiring that a developer understand the configuration as well as the code in order to change it.

Another drawback is that some IDEs might not be able to accurately analyze or refactor code when configuration is "invisible" to it. Some IDEs mitigate this problem by providing explicit support for various frameworks. Additionally, some frameworks provide configuration using the programming language itself, allowing refactoring directly. Other frameworks, such as the Grok web framework, introspect the code and use convention over configuration as an alternative form of deducing configuration information. For example, if a Model and View class were in the same module, then an instance of the View will be created with the appropriate Model instance passed into the constructor-

Criticisms

A criticism of dependency injection is that it is simply a re-branding of existing object-oriented design concepts. The examples typically cited (including the one above) simply show how to fix bad code, not a new programming paradigm. Offering constructors and/or setter methods that take interfaces, relieving the implementing class from having to choose an implementation, is an idea that was rooted in object-oriented programming long before Martin Fowler's article or the creation of any of the recent frameworks that champion it.

Types

Fowler identifies three ways in which an object can get a reference to an external module, according to the pattern used to provide the dependency:

- *Type 1* or *interface injection*, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.
- *Type 2* or *setter injection*, in which the dependent module exposes a setter method that the framework uses to inject the dependency.
- *Type 3* or *constructor injection*, in which the dependencies are provided through the class constructor.

It is possible for other frameworks to have other types of injection, beyond those presented above.

Chapter 8

Canonical Protocol Pattern & Domain Inventory Pattern

Canonical Protocol Pattern

Canonical Protocol is a design pattern, applied within the service-orientation design paradigm, which attempts to make services, within a service inventory, interoperable with each other by standardizing the communication protocols used by the services. This eliminates the need for bridging communication protocols when services use different communication protocols.

Rationale

Services developed by different project teams could be based on different communication mechanisms. As a result, a service inventory may end up having different sets of services, each conforming to a different set of protocols. When it comes to reusing services having different communication protocols, some sort of communication bridging mechanism is required. For example, services developed using JMS messaging protocol are incompatible with services using .NET Remoting, so in order to make use of these two types of services, some middleware technology needs to be in place that bridges the communication protocol disparity. Apart from incurring extra cost, the use of such a bridging technology adds latency and communication overhead. This makes the service less of a reusable and a recomposable resource and goes against the guidelines of the Service Composability design principle.

In order to design a service inventory where all services are interoperable with each other so that they can be composed into different solutions, the application of the Canonical Protocol pattern dictates standardizing the communication protocols used by the services. When all services are using the same communication protocol, the requirement for a bridging technology is eliminated and the communication between services is more streamlined.

Usage

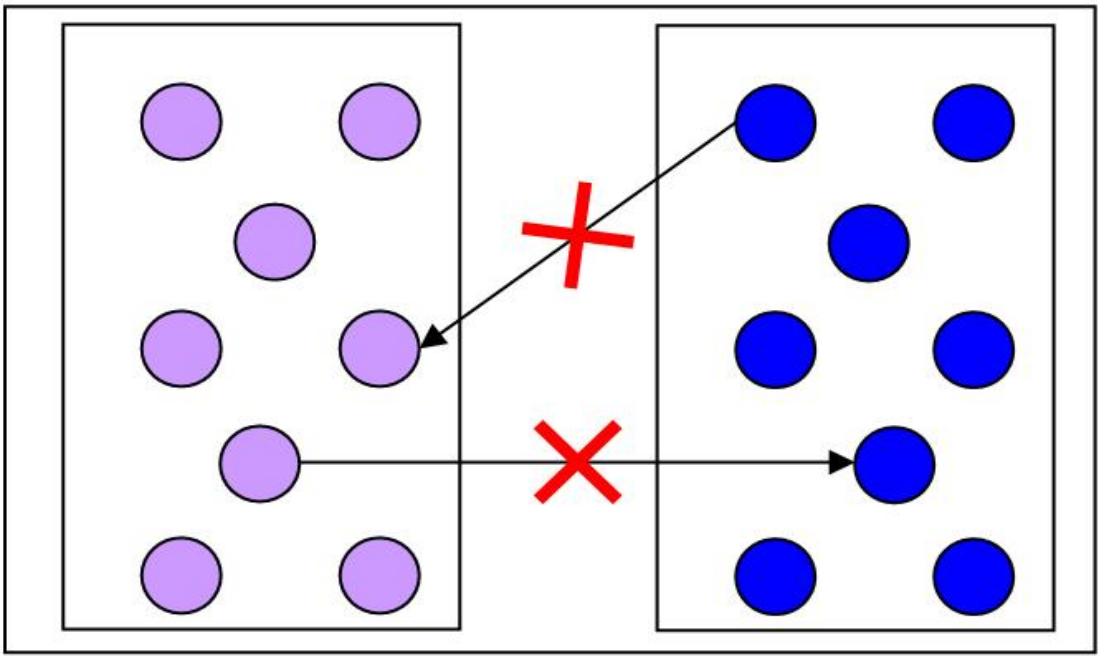


Diagram A
Services developed using different communication protocols are unable to talk to each other.

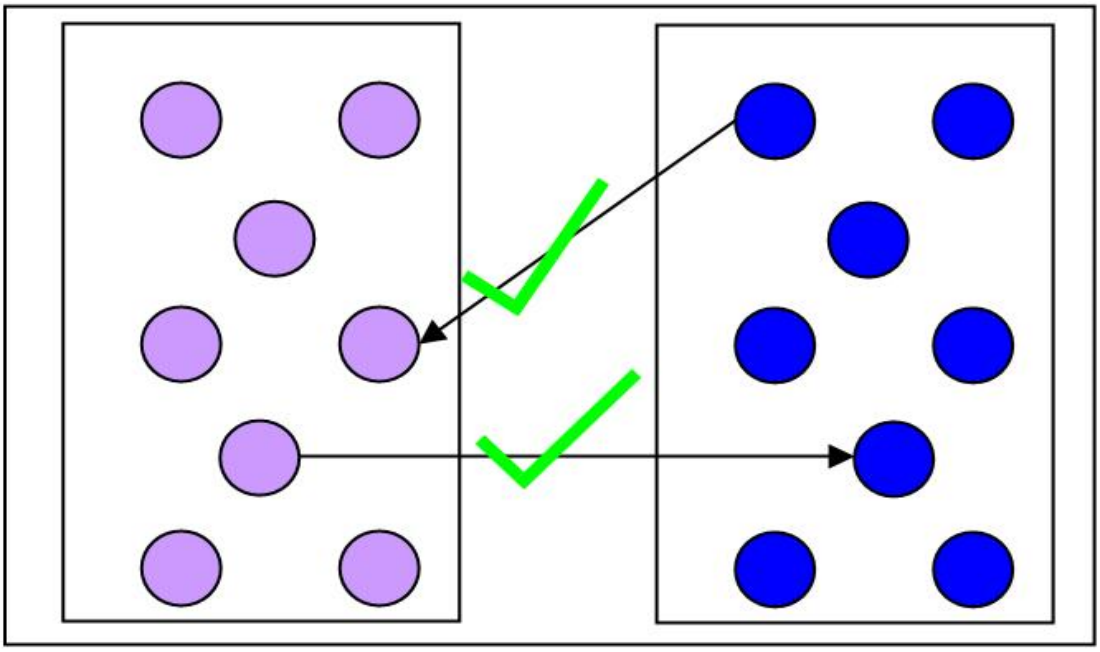


Diagram B
Services developed using the same communication protocols are able to talk to each other and hence can be used in multiple service compositions.

The application of this design pattern requires choosing a technology architecture that provides a common communication framework so that all services in an inventory can communicate with each other using the same communication protocol. This depends upon how the services within a service inventory are going to be used. If the services are only going to be part of service compositions that always use a particular communication protocol (because of efficiency and security reasons), then all the services within that service inventory can be built upon such a communication protocol even if it is not the most widely used protocol.

The Canonical Protocol pattern by Thomas Erl answers the questions, "*How can services be designed to avoid protocol bridging*?" The problem is that services that support different communication technologies compromise interoperability, limit the quantity of potential consumers, and introduce the need for undesirable protocol bridging measures. The solution is for the architecture to establishe a single communications technology as the sole or primary medium by which services can interact. Therefore, the communication protocols (including protocol versions) used within a service inventory boundary are standardized for all services.

One of the most mature and widely used communication mechanisms is provided by the Web services framework. Further to choosing a communication framework, the actual message protocols also need to be standardized upon. For example, whether web services are built using HTTP over SOAP or by simply using RESTful services. Similarly, when standardizing on SOAP based web services, the specific version of SOAP protocol needs to be agreed upon as well i.e. SOAP v 1.1 or SOAP v 1.2.

Considerations

In order to standardize on a communication protocol, the features of the protocol need to be compared against the service interaction requirements including security, efficiency and transaction support. In case of web services, for example, if a service composition requires explicit transaction support, then SOAP over HTTP would be a better choice than using RESTful services.

In some cases, depending upon the technology used to build the service, it may be possible to support two different set of protocols in order to make the service accessible to different types of service consumers(Dual Protocols design pattern). For example, using WCF, the same service can be configured to use HTTP and TCP/IP protocols at the same time.

When choosing a communication framework, the maturity, scalability and any licensing costs need to be taken into account as building services using a protocol that is going to become obsolete in the near future will impact the reusability of such services and would require considerable time and efforts in order to redesign the service.

Domain Inventory Pattern

Domain Inventory is a design pattern, applied within the service-orientation design paradigm, whose application enables creating pools of services, which correspond to different segments of the enterprise, instead of creating a single enterprise-wide pool of services. This design pattern is usually applied when it is not possible to create a single inventory of services for whole of the enterprise by following the same design standards across the different segments of the enterprise. The Domain Inventory Design pattern by Thomas Erl asks, "How can services be delivered to maximize recomposition when enterprise-wide standardization is not possible?" and is discussed as part of this podcast.

Rationale

As per the guidelines of the Enterprise Inventory design pattern, it is beneficial to create a single inventory that spans whole of the enterprise as it results in services that are more standardized, interoperable and easily composable. However, there may be situations when a single enterprise-wide inventory cannot be created. This could be because of a number of reasons including:

- management issues e.g. who will own the services and who will be responsible for their maintenance?
- the organization is spread across different geographic locations.
- different segments of the organization are supported by different IT departments and the technologies used are not the same.
- some segments of the organization might not be ready for transition towards service-orientation.
- a pilot project needs to be undertaken just to ascertain the effectiveness of SOA.
- as per the guidelines of the Standardized Service Contract, it may be very difficult to create standardized data models across the enterprise.
- cultural issues e.g. IT managers not willing to give up control they have over they way different projects are developed.

Considering the above mentioned factors, it is rather more practical to build smaller groups of services whereby the scope of a group relates to a well defined domain boundary within the enterprise. This is exactly what is advocated by the Domain Inventory design pattern. By limiting the scope of a service inventory, it becomes easier to develop and manage a group of related services.

Usage

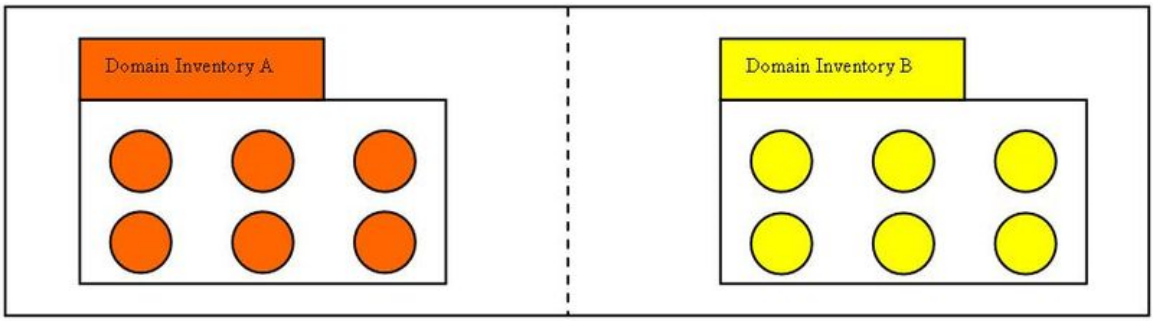


Diagram A
An enterprise consisting of two domain inventories. Services in each inventory are standardized independently according to the established boundary of the domain.

In order to apply this design pattern, a well defined boundary needs to be established inside the enterprise that would usually correspond to a particular business area of the enterprise. For example, sales department, customer services department, etc. It is important that any domains created relate to the business domains as it helps to keep the service inventory in sync with the business models as they evolve over time. Having established a well defined boundary, the next step is to create a set of design standards that would regulate the extent to which the service-orientation design principles would be applied and any other related conventions, rules and restrictions e.g. how to create the data models, how to name the service functions, etc. By having these design standards in place, standardized set of services can be developed that are specifically attuned to work within the limitations of the respective organizational segment. As the services are standardized, they can be easily composed without the requirement of any bridging mechanisms.

Considerations

If the established boundary of a domain does not correspond to an actual business domain then it might prove difficult to maintain such an inventory of services because of the managerial cross-over. Each domain inventory now corresponds to a specific set of standards that may be different from rest of the domain inventories. As a result, when it comes to composing a solution out of services that belong to different domain inventories, some sort of transformation mechanisms may be required in order for the messages to be sent between different service inventories. For example, services within domain inventory A may be using XML schemas that are less granular as compared to the schemas used by the services belonging to domain inventory B. Design patterns like the Data Model Transformation, the Data Format Transformation and the Protocol Bridging design patterns can be applied in order to address the different transformation requirements.
Another important factor is that as different domain inventories are being built by different project teams, there is a higher chance of developing services with duplicate

functionality as each team is unaware of the requirements of the other business processes that are being automa

Chapter 9

Event-Driven SOA

Event-driven SOA is a form of service-oriented architecture (SOA), combining the intelligence and proactiveness of event-driven architecture with the organizational capabilities found in service offerings. Before event-driven SOA, the typical SOA platform orchestrated services centrally, through pre-defined business processes, assuming that what should have already been triggered is defined in a business process. This older approach (sometimes called SOA 1.0) does not account for events that occur across, or outside of, specific business processes. Thus complex events, in which a pattern of activities—both random and scheduled—should trigger a set of services is not accounted for in traditional SOA 1.0 architecture.

SOA 2.0

SOA 2.0 architecture, ("event-driven SOA"), lets business users monitor, analyze, and enrich events to make the connections among disparate events that do not at first appear to be intuitively obvious. This makes these enriched events visible to others, especially business analysts or marketing directors, and also allows the SOA 2.0 system to possibly automate actions to take to address some unique pattern.

SOA 2.0 is the ability to create high-level business events from numerous low-level system events. Events are created by filtering real-time data (from middleware, applications, databases, and Web services, for example) and infusing it with defining detail such as dependencies or causal relationships discovered by correlating other events.

If it's clear, through the enriched events that are produced by an SOA 2.0 environment, that customer shopping cart abandonment rate has escalated in the last few days, a notification to the marketing department could initiate research into what competitors have done to cause customers to buy products elsewhere. Was there a common product in most shopping carts? If so, what are the prices that are being offered by the competition?

In practice, this relationship of streamed events is processed through a causal vector engine, which performs a lookup based on recently viewed events and assigns a causal vector to an event if a relationship is discovered. If A causes B, the causal vector engine checks if B’s causal vector rule index contains a reference to A. The engine may handle events for different transactions simultaneously, perhaps in a different order than they occurred.

Unlike sequential or procedural systems (in which clients must poll for change requests), event-driven SOA allows systems and components to respond dynamically, in real time, as events occur. SOA 2.0 complements and extends SOA 1.0 by introducing long-running processing capabilities.

Long running processing capability enables the architecture to collect various asynchronous events over a long period of time and correlate these events into causal relationships. SOA 2.0 event patterns can be designed and implemented to look for event relationships that span days, weeks, or months; and when certain criteria are met, trigger a business process to address the event pattern.

SOA 2.0 event-driven programming is structured around the concept of decoupled relationships between event producers and event consumers: an event consumer doesn't care where or why an event occurs; rather, it's concerned that it will be invoked when the event has occurred. Systems and applications that separate event producers from event consumers typically rely on an event dispatcher, or channel. This channel contains an event queue that acts as an intermediary between event producers and event handlers.

Prototypical SOA 2.0 paradigm

The prototypical SOA 2.0 paradigm contains four essential elements: (1) multiple low-level system events that, separately, do not appear to have any relationship, but through pattern detection by comparing these many events some unusual or less obvious correlation becomes clear; (2) some amount of data enrichment by infusion of related information to each event to more clearly illustrate how the many events are related; (3) a trigger condition which when not met, the business-level event is not created, but when the trigger condition is met, the higher-level business event is created; and (4) some human or automated process that is invoked when the trigger event is reached.

SOA 2.0 Web Services can be composed in two ways: orchestration and choreography. In orchestration, a central process takes control over the involved web services and coordinates the execution of different operations on the web services involved in the operation. The involved SOA 2.0 services do not know (and do not need to know) that they are part of a composition or a higher business process. Only the central coordinator of the orchestration knows this, so the orchestration is centralized with explicit definitions of operations and the order of invocation of SOA 2.0 services.

Choreography on the other hand does not rely on a central coordinator. Rather, each SOA 2.0 service involved in the choreography knows exactly when to execute its operations (based on defined trigger criteria) and whom to interact with. Choreography is a collaborative effort focused on exchange of messages. All participants of the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges.

BPEL follows the orchestration paradigm. Choreography is covered by other standards, such as WSCI (Web Services Choreography Interface) and (Web Services Choreography Description Language).

Multiple low-level system events

Causal relationships are inherent in the world around us and are intrinsic to our decision making. The human intelligence processes and gathers these relationships faster than current artificial computational capability can. One of the fundamental obstacles in artificial intelligence is the absence of an automated ability to relate events together as when a human uses human intuition.

Using a Causal Vector Engine, the perception of causality can be enhanced under appropriate spatiotemporal conditions based on structural and temporal rules written into the engine. Perception of complex causal semantics, such as additive, mediated, and bidirectional causalities need to be coded so that the engine can distinguish between events that are related and those that only appear to be related but, in fact, are not.

The engine uses preponderant causal vector rate-of-change propagation to code the relationship among the events and establishes a partial order in which it validates the causality perceived between multiple occurrences. The engine plays and replays the event sequence in different temporal orderd to infer what could be related topological connections and compares these replays to rules preprogrammed by an analyst.

Multiple low-level system events are processed by the Causal Vector Engine and compared against these rules to trigger higher-level Business Events. It does this through a Causality Vector Engine (CVE) console application which displays events in real-time to business analysts. Where streams of events can be observed as they occur, much like a stock ticker, the CVE console app has several windows that list the same events in different contexts, so the business analysts can see what the CVE is doing with the relationships between them.

The Sequential window shows events in date-timestamp order, one or more other windows in various orders as the CVE works through the list of rules and creates implied relationships between the events. Various buttons and controls exist in the console application that enable the business analysts to create relationships between events on-the-fly and define rules that respond to these relationships.

Business analysts can infuse additional defining detail through an SQL query statement attached to a rule or event context. The CVE app works much like a modern day stock trading application that mutual funds managers use to manage risk.

Data enrichment

Most ESB implementations contain a facility called "mediation". For example, mediation flows are part of the WebSphere enterprise service bus intercept. Mule also supports

mediation flows. Mediation flows modify messages that are passed between existing services and clients that use those services. A mediation flow mediates or intervenes to provide functions, such as message logging, data transformation, and routing.

As messages pass through the ESB, the ESB enriches the messages destined for a channel that is monitoring for a high-level business event. That is, for each message, the ESB may query a database to obtain additional information about some data entity within the message. For example, based on Customer ID, the ESB mediation flow could get the zip code that the customer resides in. Or, based on IP address of the originating request by the end-user, the ESB mediation flow could lookup what country, state or county that IP address is in.

These examples represent data enrichment, the concept of adding additional value to existing data, based on the intent of the high-level business event to eventually be triggered.

Mediation flows

An ESB mediation flow is one of the component types in a Service Component Architecture (SCA). Like any SCA component, the program accesses a mediation flow through exports that it provides, and the mediation flow forwards messages to other external services via imports. Special kinds of imports and exports for JMS, called JMS bindings, enable developers to specify the binding configuration and write data handling code. The mediation flow consists of a series of mediation primitives that manipulate messages as they flow through the bus.

Once the developers have coded the custom binding for both export and import, they can start to focus on the mediation flow component. In the WebSphere Integration Developer assembly editor, this is done by the JMS Custom Binding Mediation Component where each operation on the flow component's interface is represented by a request and a response.

Service Data Objects (SDO) framework provides a unified framework for data application development. With SDO, developers do not need to be familiar with any specific API in order to access and utilize data. Through SDO, developers simply work with data from multiple data sources, such as relational databases, entity EJB components, XML pages, Web services, the Service Component Architecture, and JavaServer Pages pages.

Mediation flows are entirely independent from the bindings that are used in the imports and exports. In fact, the purpose of having a conversion into an SDO DataObject instance outside of the flow implementation is because mediation flows can then be built without knowledge of the protocol and format with which messages are sent to and from the mediation module.

Business-level trigger condition

A business-level trigger condition enables the SOA 2.0 architecture to establish real-time customer intelligence, marketing automation and customer loyalty solutions, among other features. Business objects model real-world entities in the architecture such as customers, accounts, loans, and travel itineraries. When the state of one of these objects changes, and a monitoring agent notices this change is significant (when compared to the list of criteria to monitor), an event is created and passed to other monitoring agents.

For example, the detection of an actual business problem or opportunity could lead to increased revenue. If a customer cancels an order, extra manufacturing capacity could reduce the profitability of the production run. A SOA 2.0 event could notify marketing department to create a special sales campaign that would resell the excess capacity, thereby recapturing the original profitable cost-per-unit.

Automatic monitoring of events in operational business process activities as processes execute to see if any immediate action needs to be taken either inside or outside the enterprise. These monitoring agents continually test for specific business conditions and changes in business operations. If necessary, the agents alert people, make recommendations, send messages to other applications or invoke whole business processes when such conditions or changes occur.

Resulting business process

A triggered business process should directly support revenue growth with cost containment, responsiveness to business conditions, or ability to pursue new market opportunities. Resulting business processes could also measure operational progress toward achieving goals, control operational costs by communicating just what is needed to just who needs to know, or report performance status of key processes to key decision makers.

SOA 2.0 Conceptual Examples

Abandoned Shopping Cart

For example, you could construct a CRM event from an "abandoned shopping cart" message (parsing the transaction, customer ID, and time), using other filters to extract the value of goods in the cart and tapping the correlation capabilities of the system to add causal indicators such as whether the commerce site was suffering performance problems. Your CRM event might also include customer value or rank from the customer database.

Engineering Defect

For another example, based on the types of independent service calls received, the SOA 2.0 platform could identify a product defect by detecting the underlying pattern of the

separate complaints, then triggering an alert to engineering or production of the possible defect.jou kale ma is sinan

Real-time Electricity Market

Example 3: A potential use of event-driven SOA could be a virtual electricity market where home clothes dryers can bid on the price of the electricity they use in a real-time market pricing system. The real-time market price and control system could turn home electricity customers into active participants in managing the power grid and their monthly utility bills. Customers can set limits on how much they would pay for electricity to run a clothes dryer, for example, and electricity providers willing to transmit power at that price would be alerted over the grid and could sell the electricity to the dryer.

On one side, consumer devices can bid for power based on how much the owner of the device were willing to pay, set ahead of time by the consumer. On the other side, suppliers can enter bids automatically from their electricity generators, based on how much it would cost to start up and run the generators. Further, the electricity suppliers could perform real-time market analysis to determine return-on-investment for optimizing profitability or reducing end-user cost of goods.

Event-driven SOA software could allow homeowners to customize many different types of electricity devices found within their home to a desired level of comfort or economy. The event-driven software could also automatically respond to changing electricity prices, in as little as five-minute intervals. For example, to reduce the home owner's electricity usage in peak periods (when electricity is most expensive), the software could automatically lower the target temperature of the thermostat on the central heating system (in winter) or raise the target temperature of the thermostat on the central cooling system (in summer).

The event-driven SOA software could shut off the heating element of water heaters to the pre-set response limits established by individual homeowners. For example, if the market price of electricity for a given hour exceeded the home owner's limit, the home owner could plan to go without recharging the water's hot temperature for that hour, when prices were high, and opt to delay the hot water temperature recharge to the next hour when electricity market prices might be lower.

All this criteria would be managed through the home owner's personal computer with internet connection, programming the various devices around the home to consume electricity only when the management software approves of the consumption. The savings represented by this technique, and enabled by event-driven SOA, is like improving the gas mileage in your vehicle. It makes your home energy use more efficient by enabling the consumption of electricity when the real-time prices are lower and inhibiting the consumption of electricity when real-time prices are higher.

SOA 2.0 Implementations

One mechanism that can be used from most SOA 1.0 Enterprise Service Bus implementations is the publish/subscribe facility. By implementing ESB functionality as Pub/Sub messages, no advanced knowledge of system events is needed to create SOA 2.0 message patterns. After an enterprise has implemented many Publish functions, SOA middleware analysts can set about the task of strategizing which of the available Publish messages could be assembled into a unique pattern to detect an SOA 2.0-enriched trigger.

CVE mechanics are implemented simply, with an expandable view of SQL constructs written in stored procedures. If A causes B, and causality must occur within N number of transactions, then SQL ORDER BY timestamp clause creates a result set that increments a counter of all transactions that occurred within a timeframe, N number of matching B to occurrence A transactions. The creation of additional stored procedures is accomplished through the CVE console application or by using any standard database developer's toolkit.

Chapter 10

Entity Abstraction Pattern & Event-Driven Messaging

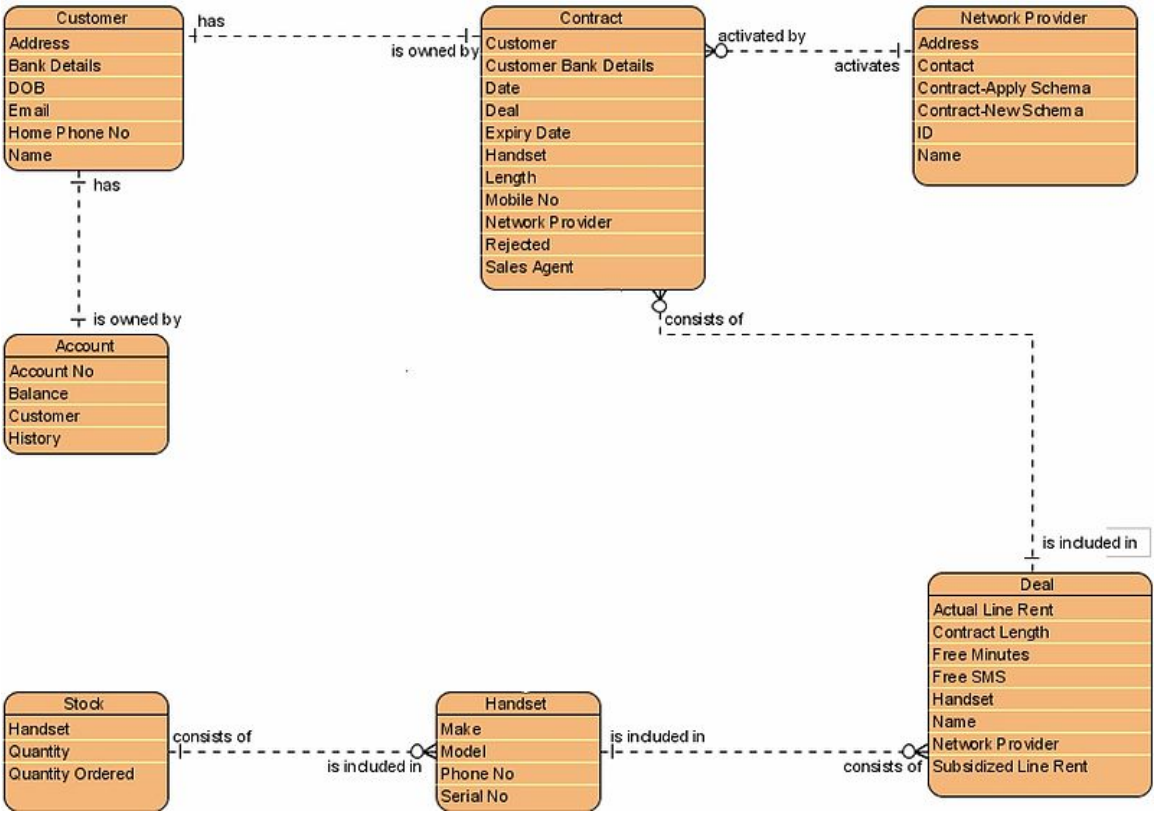
Entity Abstraction Pattern

Entity Abstraction is a design pattern, applied within the service-orientation design paradigm which provides guidelines for designing reusable services whose functional contexts are based on business entities.

Rationale

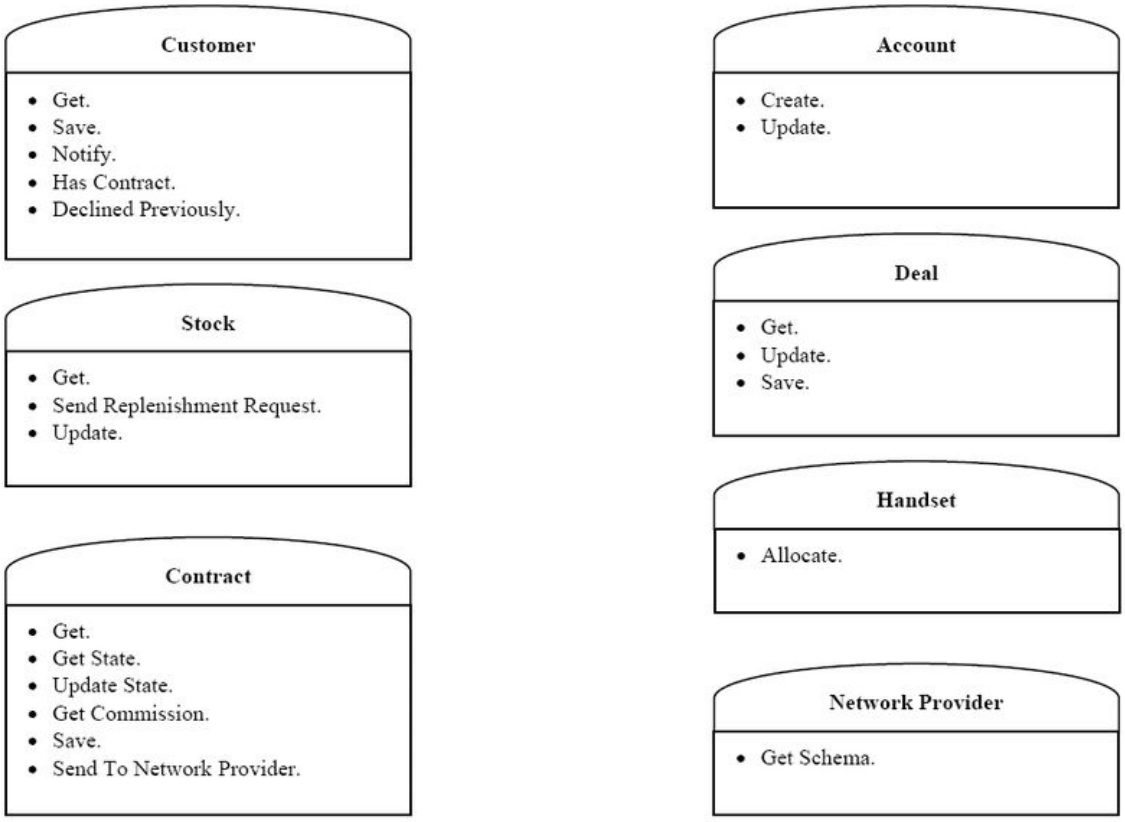
The automation of a business process involves the analysis of the business domain and then designing solution logic that represents the different steps within the business process. Some of these steps relate just to that specific business process while others may be of use to other business processes as well. Part of this reusable logic pertains to the business entities that usually remains the same when compared to the rules and processing steps that may change in future. If services are designed that contain both process-specific logic and entity-specific logic, the chances of reusing the same entity-specific logic, from another business process, become somewhat negligible. On the other hand, if this kind of logic is split up into a separate container i.e. a service, then any new business processes, which make use of the same business entity, can reuse this logic. Apart from the reusability problem, in order to address the change in the behavior of a business entity, updating the entrenched entity related logic across multiple business processes requires extra efforts and makes the maintenance of such services a complex task.
In response to the aforementioned issues, the Entity Abstraction pattern advocates that logic that relates to the processing of business entities be separated from the process-specific single purpose logic and designed as independent logic that has no knowledge of the overall business process in which such logic is being utilized.

Usage



ERD

An ERD showing the relationships between different business entities and their attributes.



Entity Services

Entity specific logic is identified from the above ERD and abstracted away in relevant functional contexts that constitute the entity services.

The separation of reusable entity-specific logic requires identification of such logic before the actual services are designed. This requires a top-down service-oriented analysis and design service delivery process. During the analysis phase, different types of actions that are performed on and by business entities are identified and placed in a relevant process-neutral functional context that forms the basis for entity services. These entity services specifically contain functionality, including CRUD functions, that is only relevant to the physical or logical business entities, represented by their corresponding functional context. Apart from the identification of different actions, it is also important to discover any relationships between the business entities whether part of the current business process or not. By looking across different business processes, the entity service can be packed with extra functionality that may be required by other business processes. An important source of information for identifying such relationships are the ERDs as they physically display the relationships between different business entities and also identify the different attributes of entities that form the basis of the relationships among the entities. The resulting set of services represents the entity layer, which is one of the layers as advocated by the Service Layers design pattern. The application of the Entity Abstraction design pattern can be viewed as a specialized implementation of the Agnostic Context design pattern as the Agnostic Context pattern advocates separating process-neutral logic from the process-specific logic, however, in case of Entity

Abstraction pattern it is the entity-specific process-neutral logic that is being separated. The process of identifying entity services from business entities might not always result in a one-to-one mapping as on some occasions it might be logical to combine two entities into one because on their own they do not represent a significant part of functionality being performed on the entities.

Considerations

The application of this design pattern requires adopting top-down service-oriented analysis and design service delivery process which may not be suitable for organizations that have limited resources, both in terms of time and man-power. Another important aspect is that once the entity-specific logic has been abstracted away in entity services, the dependency on such entity services increases as they become the sole point of contact for utilization functionality that is of interest to multiple business processes. Consequently, it remain very important to have strict governance mechanisms in place so that a change in the functionality of an entity service does not result in behavior that is not expected by the service consumers that have already formed dependencies on such entity services.

Event-Driven Messaging

The **Event-Driven Messaging** is a design pattern, applied within the service-orientation design paradigm in order to enable the service consumers, which are interested in events that occur within the periphery of a service provider, to get notifications about these events as and when they occur without resorting to the traditional inefficient polling based mechanism.

Rationale

The interaction between a service consumer and a service provider is normally initiated by the service consumer as it needs to respond to an event that occurs within the boundary of the service consumer itself e.g. requiring some data from an external resource (i.e. the service provider) in order to perform a calculation whose results need to be relayed back to a user interface in response to an action performed by the user. However, there are situations where the service consumer needs to wait for the occurrence of an event within the boundary of the service provider itself. Under these circumstances, the service consumer somehow needs to be informed of the event as and when it happens. One way is to program the service consumer to poll the service provider with regular intervals so that it can check if the event happened or not. This approach not only manifests inefficiency but also behavioral unpredictability. Inefficiency because the service consumer and the service provider are engaged in unproductive interactions and unreliable because it might be that the event actually happened more than once before the

service consumer could poll the service consumer, thereby missing the previous events and their related data. Apart from these problems, such a technique also introduces latency as the interval with which the service consumer performs the polling is fixed and, therefore, it would only fetch the event data at that time and not when the event actually occurred. This whole scenario deteriorates even further if multiple service consumers are dependent on a particular service provider.

In order to tackle this problem, the Event-Driven Messaging design pattern suggests a publisher-subscriber communication mechanism that ensures timely notification of event related data to the service consumer, thereby eliminating the inefficiencies linked with the traditional polling based communication mechanism.

Usage

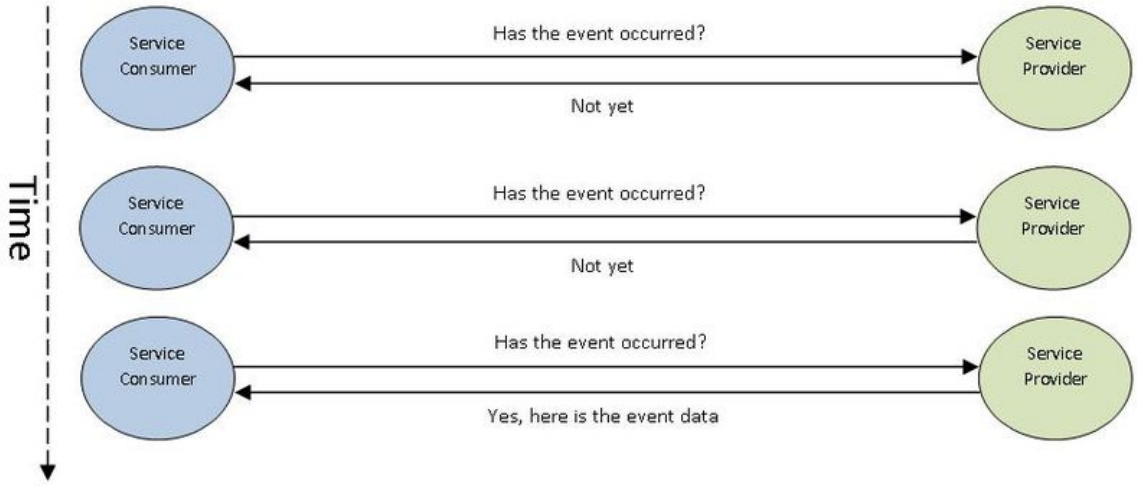


Diagram A
In order to find out if a particular event has occurred or not, the service consumer polls the service provider with regular intervals, which results in inefficient service interactions.

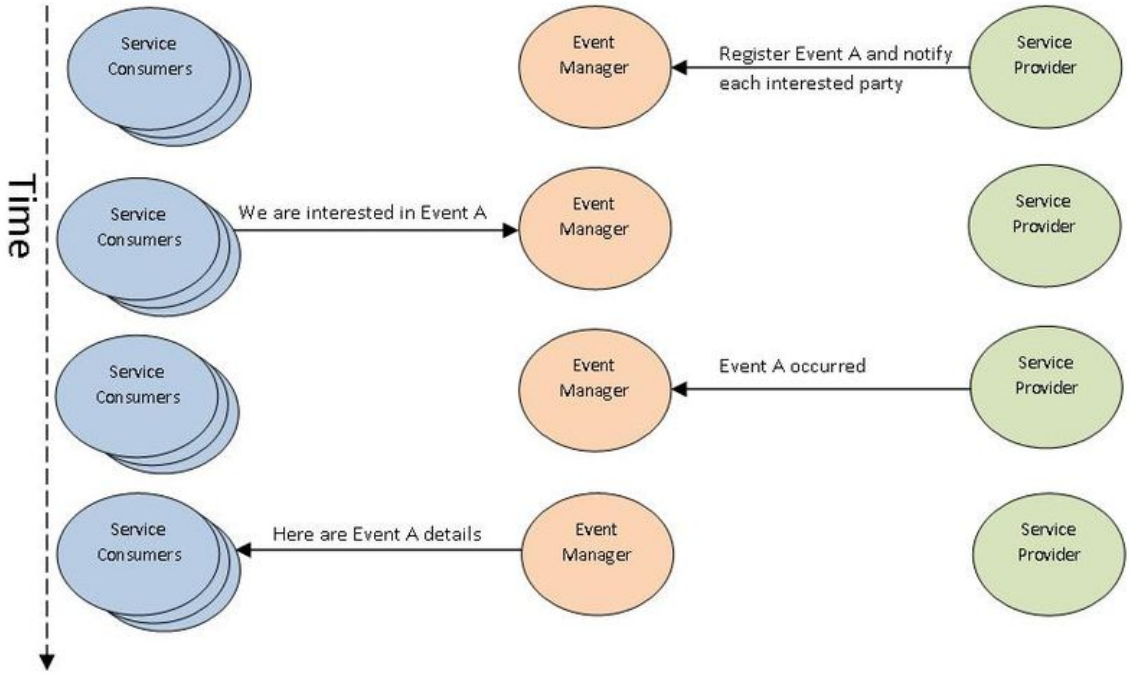


Diagram B
The event manager automatically notifies all the interested service consumers about the occurrence of a particular event the moment it actually happens.

The application of the Event-Driven Messaging design pattern requires an event manager with whom the service provider registers its events. The service consumers then register their interest in few or all of the advertised events. Upon the occurrence of an event, the service provider informs the event manager that then notifies all of the registered service consumers instantly. This communication mechanism shares its roots with the Observer pattern applied traditionally within the object-oriented world. This design pattern also borrows some concepts from the Event-Driven Architecture as the fundamental rationale behind this design pattern is responding to events.

The actual implementation of such a publisher-subscriber based communication mechanism requires architectural extensions in order to provide such a complex message tracking and forwarding mechanism. A mature ESB product should normally be able to provide such functionality. The application of this pattern helps to further decouple the service consumers from the service providers and increases the overall reliability of a service composition.

Considerations

The application of this pattern is dependent upon the existence of underlying platform extensions, which if not already present, would incur extra cost and therefore would impact the IT budget. It should also be noted that the publisher-subscriber model is based on asynchronous messaging, so the transmission of a message from the event manager

can take place at any time, which could mean that if the event manager broadcasts an event notification message then it's not necessary that the service consumer would be online to receive it. Therefore, the application of this design pattern does not solve the unavailability problems. However, this could be addressed by further applying the Asynchronous Queuing and the Reliable Messaging design patterns that guarantee that a transmitted message is always received by the intended receiver along with acknowledgment messages.

The introduction of architectural extensions would affect the current service inventory architecture and the way service compositions are designed, therefore, also affecting the service composition architectures.

Chapter 11

Canonical Schema Pattern & Enterprise Inventory

Canonical Schema Pattern

In software engineering, **Canonical Schema** is a design pattern, applied within the service-orientation design paradigm, which aims to reduce the need for performing data model transformation when services exchange messages that reference the same data model.

Rationale

The interaction between services often requires exchanging business documents. In order for a service consumer to send data (related to a particular business entity e.g. a purchase order), it needs to know the structure of the data i.e. the data model. For this, the service provider publishes the structure of the data that it expects within the incoming message from the service consumer. In case of services being implemented as web services, this would be the XML schema document. Once the service consumer knows the required data model, it can structure the data accordingly. However, under some conditions it may be possible that the service consumer already possesses the required data, which relates to a particular business document, but the data does not conform to the data model as specified by the service provider. This disparity among the data models results in the requirement of data model transformation so that the message is transformed into the required structure as dictated by the service provider. Building upon the aforementioned example, it is entirely possible that, after processing the received business document, the service provider sends back the processed document to the service consumer that once again performs the data model transformation to convert the processed business document back to the data model that it uses within its logic to represent the business document.
This runtime data model transformation adds processing overhead and complicates the design of service compositions. In order to avoid the need for data model transformation, the Canonical Schema pattern dictates the use of standardized data models for those business documents that are commonly processed by the services in a service inventory.

Usage

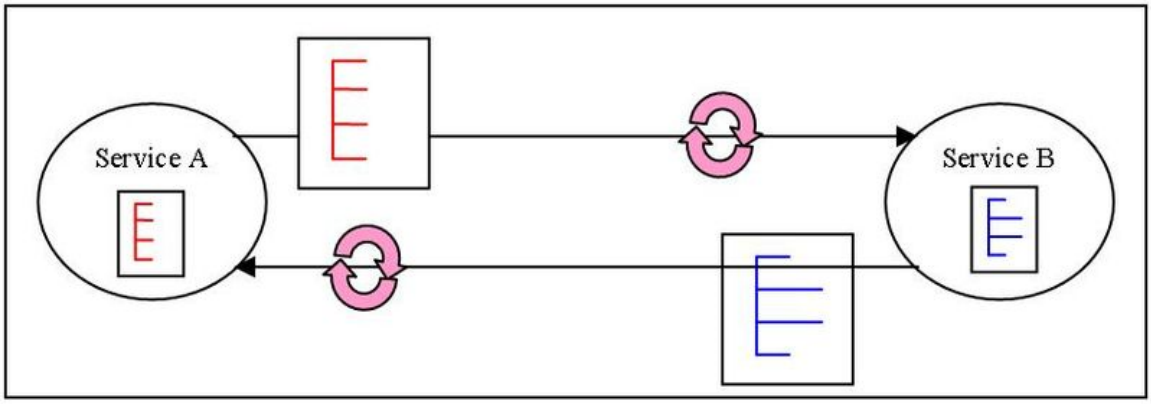


Diagram A
Service A is using a different data model as compared to Service B for the same business document. When messages are exchanged, runtime data model transformation needs to be performed.

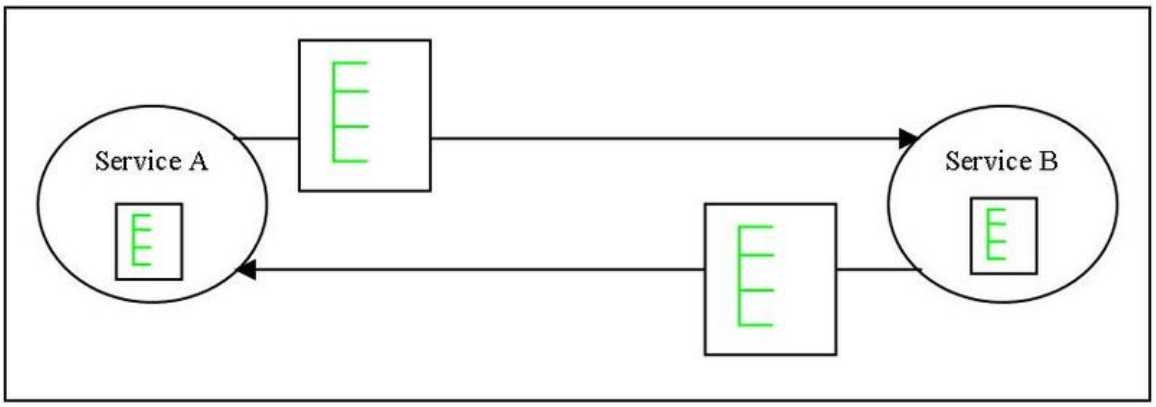


Diagram A
Both services are using the same data model for representing a particular business document. AS a result, no data model transformation is required when messages are exchanged.

This design pattern is fully supported by the application of the Standardized Service Contract design principle. The Standardized Service Contract design principle advocates that the service contracts be based on standardized data models. This is achieved by performing an analysis of the service inventory blueprint in order to find out the commonly occurring business documents that are exchanged between services. These business documents are then modeled in a standardized manner. For example, in case of web services, the business documents are modeled as XML schemas. Once a standardized data representation layer exists in a service inventory, different service contracts can make use of the same data models if they need to exchange the same business documents. This eliminates the need for any data model transformation and reduces the processing overhead associated with the data model transformation. It also

increases the reusability potential of a service as now the service can be consumed without requiring any custom data model transformation logic. In a way, the application of the Canonical Schema pattern reduces the need for the application of the Data Model Transformation design pattern.

Considerations

The application of this design pattern requires design standards in place that make the use of standardized data models mandatory, as the mere creation of data models does not guarantee their use. Although simple in principle but difficult to enforce as it needs commitment from different project teams which may entail extra efforts, on part of each team, in terms of designing solutions that accommodate standardized data models.
On some occasions, either because of the sheer size of the organization or because of the resistance from different segments of the enterprise, the Canonical Schema design pattern may need to be applied within a particular domain inventory, created by the application of the Domain Inventory design pattern.
The schemas need to be designed separately than the service contract design so that there is no dependency between them.

Enterprise Inventory

In the domain of the service-orientation design paradigm, the **Enterprise Inventory** is a design pattern by Thomas Erl that answers the question, "How can services be delivered to maximize recomposition?"; the application of this pattern results in a standardized enterprise-wide service inventory that fosters repeated service composition.

Rationale

An SOA adoption usually results in multiple set of services built by various teams as part of automating different business processes spread across diverse departmental boundaries. Each project team might be following different set of standards and implementation architectures (for developing services) that are more relevant to the team's short-term goals and requirements. Although the built services might provide enough opportunities for reuse and recomposition within the same business domain, however, when it comes to composing services from different business domains, there tends to be an impedance mismatch requiring some sort of bridging or transformation logic in between these services. In a way, such services result in standalone solutions that defy the basic characteristics of service-orientation i.e. by not being enterprise-centric. Quite often such distributed service delivery efforts result in the development of redundant services as project teams are not aware of each other’s requirements.
In order to foster an environment whereby all the services conform to a single set of standards, the Enterprise Inventory design pattern is applied that results in a common

service inventory that is based on a service-oriented enterprise architecture. This automatically eliminates any redundancy and paves the way for maximum recomposition of services which could mean development of new solutions with reduced efforts and time.

Usage

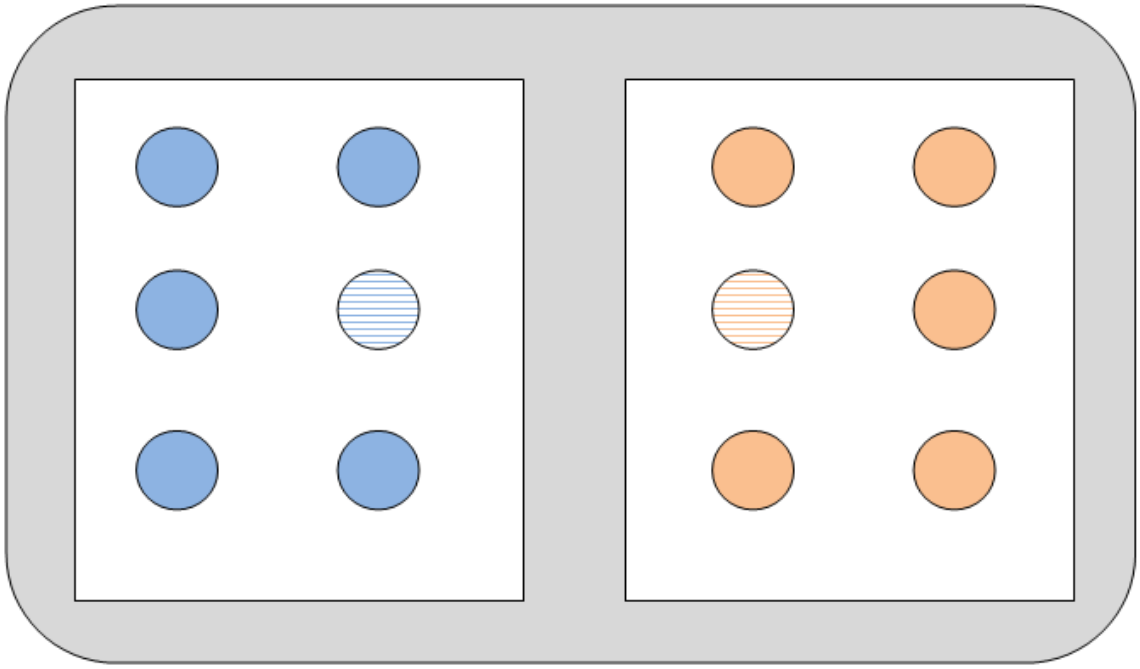


Diagram A
Two individual service inventories belonging to two different organizational departments built using different standards and based on different implementation architectures. The shaded services represent redundant services.

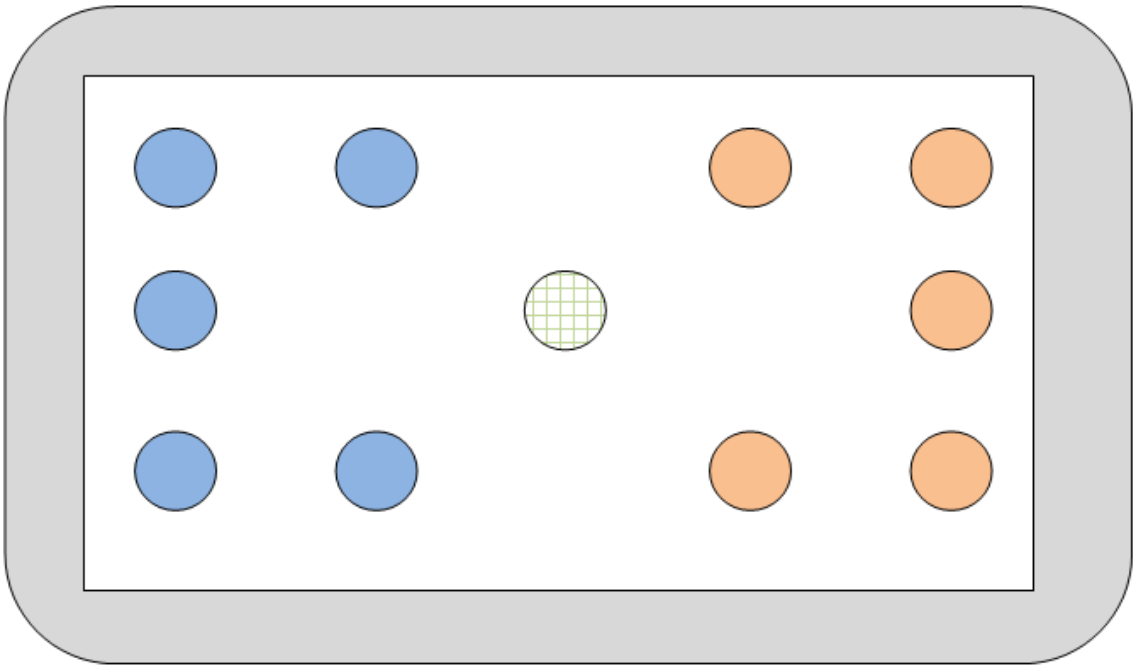


Diagram B
A single enterprise-wide inventory built around a common set of standards providing maximum recomposition opportunities and eliminating any redundancies.

As depicted in Diagram A, the services belonging to two different departments, although belonging to the same organization, contain architectural incompatibilities as well as redundancy. An inter-department service composition is not possible until and unless there is some sort of transformation logic introduced in between these services. However, in Diagram B, the application of Enterprise Inventory pattern creates a standardized service inventory that is inherently interoperable.
In order to be sure that all the services that are being built follow the same design standards, as part of the application of this design pattern, a service inventory blueprint needs to be created. Such a blueprint only consists of candidate services containing candidate capabilities. By coming up with such a service inventory blueprint, the overall scope of the potential enterprise-wide service inventory is established. This usually requires the application of the top-down service-oriented analysis.
Apart from the creation of an enterprise inventory, the application of this pattern also requires that a technology architecture based on the current enterprise architecture needs to be documented as well so that the services could be built within the boundaries of such an architecture. Doing so guarantees service interoperability as well as behavioral predictability.

Considerations

As the application of the Enterprise Inventory design pattern requires upfront analysis, it is more suited towards organizations that have IT systems with well established procedures and documentation in place. If it is a fresh SOA initiative then creation of an

enterprise-wide inventory would be rather easy and straightforward. Being an enterprise-wide initiative, it would be rather difficult for existing services to be adapted to the new design standards and would incur financial burden as well as require considerable amount of time.

In some circumstances, it might not be feasible to create a single enterprise service inventory because of the sheer size of the organization. This would also mean that governing and maintaining such an inventory might also become impossible as there are simply too many services to govern and maintain. Last but not the least, a host of cultural issues might prevent the adoption of a common enterprise service inventory. This could include hesitance towards giving up control the way projects are developed, reluctance towards adopting design standards that restrict efficient development of projects and extra burden on service developers in terms of keeping track of enterprise-wide standards and to keep a constant check as to what other projects teams are up to in order to avoid any redundancy.

Chapter 12

Service Abstraction & Service (Systems Architecture)

Service Abstraction

Service Abstraction is a design principle that is applied within the service-orientation design paradigm so that the information published in a service contract is limited to what is required to effectively utilize the service i.e. the service contract should not contain any superfluous information that is not required for its invocation. Also that the information should be limited to the serviced contract (technical contract and the SLA) only i.e. no other document or medium should be made available to the service consumers other than the service contract that contains additional service related information.

Purpose

A service contract that contains details about what is encapsulates e.g. the logic, implementation and the technology used to build the service, may end up being used in a particular manner as now the service consumer has more knowledge about the working of the service. Under normal circumstances, the more knowledge we have, the better it is but in case of service-orientation, there is a chance that the additional information could impede the reusability of the service as the service consumer designer may streamline his design based on this knowledge, however, this would affect the evolution of the service contract as now the service consumer is indirectly coupled to the service implementation, which may need to be replaced in the future. In a way, this increases the consumer-to-contract type of coupling, which although is a positive type of coupling but having too much dependence can negatively impact the evolution of both the service provider and the service consumer.
Information hiding remains one of the key principles within object-oriented paradigm that promotes abstracting away the inner workings of a software program. A classic example would be the use of abstract classes to hide the actual method logic. The same concept is applied by the Service Abstraction principle in order to hide the unnecessary details about the working of the service with a view to ease the evolution of the service.

Application

The application of this design principle requires looking into four different types of abstractions that could be applied to a service.

Functional Abstraction

This form of abstraction is dependent upon how much of the service logic is exposed as service capabilities. An example would be of a class whereby some of its methods are private while others are public. A class would only expose those methods as public that it deems to be important to its objects, any helper methods that are not relevant to the objects are kept hidden.
A service contract which has not been subjected to this principle could be termed as a 'detailed contract' that reveals much of business rules and the validation logic. Once this principle has been applied to a fair degree, the contract could be termed as a ‘concise contract’. Further application of this design principle would result in a 'optimized contract' that maximizes the reuse potential of the service.

Technology Information Abstraction

Any information about the underlying technology used within the service would result in a low technology information abstraction as the service contract explicitly tells the service consumers how the service logic and its implementation are designed. This extra information might result in service consumers being designed in a way that specifically targets that particular implementation, thereby developing consumer-to-implementation coupling.

Logic Abstraction

The programmatic details about the service logic need to be abstracted as knowledge about how the service actually performs its functionality can result in service consumers that factor in this information and are consequently designed under these assumptions. This can seriously hamper service logic refactoring efforts and can be considered as an anti-pattern towards the application of the Service Refactoring design pattern.

Quality of Service Abstraction

This kind of abstraction relates to the details provided within the service’s accompanying SLA. It’s important to concentrate only on that kind of information that would actually help in determining the reliability and availability of the service, no other information should be included that exposes unnecessary details e.g. details about how does a service sits within the overall business process and which other services it uses for fulfilling its functionality.

The level of access control applied to a service would decide how much of technology, logic and QoS abstractions are in place. An 'open access' would provide free access to

everyone that is interested in finding out the design specifications of a service. In a 'controlled access', only authorized people are granted access and a 'no access' policy would totally deny any access to the design documents.

Considerations

Although information hiding is considered a healthy practice, however, too much of information hiding could be counter productive as it can limit the reusability level of the service. This can also result in redundant services as service designers don't have enough information about the capabilities of the service. For this, each service contract needs to be designed in a concise yet comprehensive manner so that the service's capabilities can be effectively discovered and interpreted as dictated by the Service Discoverability Principle.
The kind of information exposed in the service contract can lead to some security related concerns as well. For example, a service that propagates the details about the database in use as result of an internal error can fall a victim to an attack where the attacker makes use of the reported error details and attempts to connect to the database. This could be addressed by the application of the Message Screening and Exception Shielding design patterns.

Service (Systems Architecture)

In the context of enterprise architecture, service-orientation, and service-oriented architecture, the term **service** refers to a set of related software functionalities, together with the policies that should control its usage.

OASIS defines service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description."

Service engineering

An enterprise architecture team will develop the organization's service model first by defining the top level business functions. Once the business functions are defined, they are further sectioned into services that represent the processes and activities needed to manage the assets of the organization in their various states. One example is the separation of the business function "Manage Orders" into services such as "Create Order," "Fulfill Order," "Ship Order," "Invoice Order" and "Cancel/Update Order."

Service description/specification

A service has a description or specification. This description consists of:

1. An explicit and detailed narrative definition supported by a low-level (not implementation-specific) process model. The narrative definition is in some cases augmented by machine-readable semantic information about the service that facilitates service mediation and consistency checking of an enterprise architecture.
2. A set of performance indicators that address measures and performance parameters, such as availability (when should members of the organization be able to perform the function?), duration (how long should it take to perform the function?), rate (how often will the function be performed over a period of time?), etc.
3. A link to the organization's information model showing what information the service owns (creates, reads, updates, and deletes) and which information it references that is owned by other services.

Service implementation

A Web service provides one way of implementing the automated aspects of a given business or technical service.

Telecommunication network architectures, such as IMS, are able to provide subscribers with various Next Generation Network Services.

Services can be stateless and stateful. Stateless services can be services like data aggregation services. Stateful services are used for executing business logic.

Chapter 13

Service Autonomy Principle & Service Layers Pattern

Service Autonomy Principle

Service Autonomy is a design principle that is applied within the service-orientation design paradigm, in order to services that have maximum control over their execution environment. This increases the reliability of the service as it can execute its functionality without relying on resources over which it has little or no control.

Purpose

Service-orientation design paradigm emphasizes a lot on service reuse as dictated by the Service Reusability design principle. This means that a service, which is being heavily reused, needs to guarantee its reliability as only a reliable service stands a chance of being reusable. However, in order to provide reliability, the service needs to have maximum control over the way it performs its functionality. This requires exclusive service logic and underlying implementation resources so that the service is not dependent upon external resources over which it has no control i.e. such external resources, whether shared service logic or a shared database, may not be available when required by the service.
Although the traditional component-based software development also faces the same autonomy requirement, however, the provisioning of autonomy and reliability, in such circumstances, is left to the actual runtime environment e.g. by providing failover support or by deploying a solution on dedicated servers. On the other hand, within service-orientation, the stakes are even higher as a service-oriented solution can be composed of services that exist outside of the organizational boundary. So in this case, it's the design of the service itself that matters and the service needs to be designed in a way that it exercises maximum control over how it fulfills its functionality. The Service Autonomy principle attempts to provide guidelines for designing autonomous services so that the resulting services are more predictable and reliable.

Application

The application of this design principle requires looking into two different types of autonomy in order to increase the overall autonomy of the service.

Design-Time Autonomy

This type of autonomy refers to the independence with which the services could be evolved without impacting their service consumers. This type of autonomy is required as the service's underlying legacy resources might need an overhaul or the service's logic might need refactoring in order to make it more efficient.
The application of the Service Loose Coupling and the Service Abstraction principles helps in attaining design-time autonomy as their application results in services whose contracts are shielded from their logic and implementation and hence, the services could be redesigned without affecting their service consumers.

Runtime Autonomy

This type of autonomy refers to the extent of the control that a service has over the way its solution logic is processed by the runtime environment. The more control a service has over its runtime environment, the more predictable its behavior is. This type of autonomy is achievable by providing dedicated processing resources to the service. For example, if the service logic performs memory intensive tasks then the service could be deployed to a server that is not heavily shared by other services. Similarly, by providing locally cached copies of data, where applicable, the service's dependency on a remote shared database can be removed. As a result the overall autonomy of the service is increased.

There exists a direct relationship between the runtime autonomy and the design-time autonomy. By increasing the design-time autonomy, the ability to evolve service's implementation environment is automatically increased and hence more runtime autonomy.

Service Autonomy and Service Types

Although increasing service autonomy to the maximum extent is always desirable, however, in reality it may not be possible to design each and every service with maximum design-time and runtime autonomy. As a result, the services need to be prioritized so that their autonomy could be addressed according to their value for business. This could be done by having a look at the type of the functional context contained by a service. Services whose functional contexts are independent of any particular business process i.e. entity and utility services, are good candidates for increasing their autonomy. This is because they contain functionality that is of interest to different types of consumers. On the other hand, business process specific services i.e. task and orchestrated task services, are less reusable and are dependent upon the individual autonomy of their composed services.

Considerations

The provisioning of service autonomy may require additional infrastructure and hence it needs to be applied on as per need basis by prioritizing the services. On some occasions, services may need to be isolated and deployed in a customized and dedicated environment. This puts a lot of emphasis on making sure that the service is designed with the correct functional context as making fundamental changes to such a service might not be easy.
The autonomy of the services that encapsulate legacy resources can be hard to predict and increase. This may require additional analysis on part of the utility services as the achieved level of autonomy would depend upon the type of the functionality provided by the utility service.

Service Layers Pattern

Service Layers is a design pattern, applied within the service-orientation design paradigm, which aims to organize the services, within a service inventory, into a set of logical layers. Services that are categorized into a particular layer share functionality. This helps to reduce the conceptual overhead related to managing the service inventory, as the services belonging to the same layer address a smaller set of activities.

Rationale

Grouping services into functional layers reduces the impact of change. Most changes affect only the layer in which they're made, with few side-effects that impact other layers. This fundamentally simplifies service maintenance.

The Service Reusability principle dictates that services should be designed to maximize reuse. Similarly, the Service Composability principle advocates designing services so that they can be composed into various ways. Both principles require that a service contain only a specific type of logic e.g., either reusable or process-specific logic. Restricting each layer to a particular functionality, simplifies the design of the service.

Usage

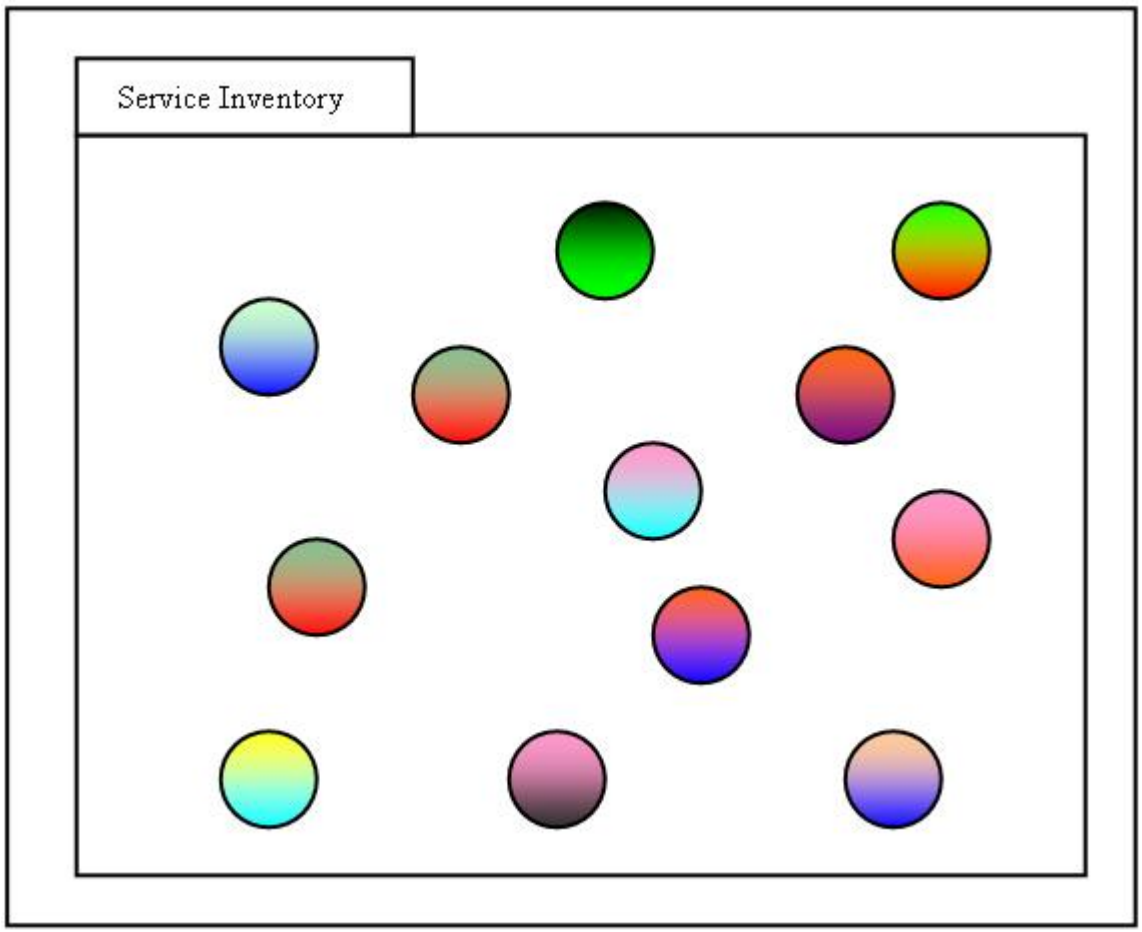


Diagram A
In the absence of any layers, services contain a mixture of different types of logic. This makes it difficult to manage these services.

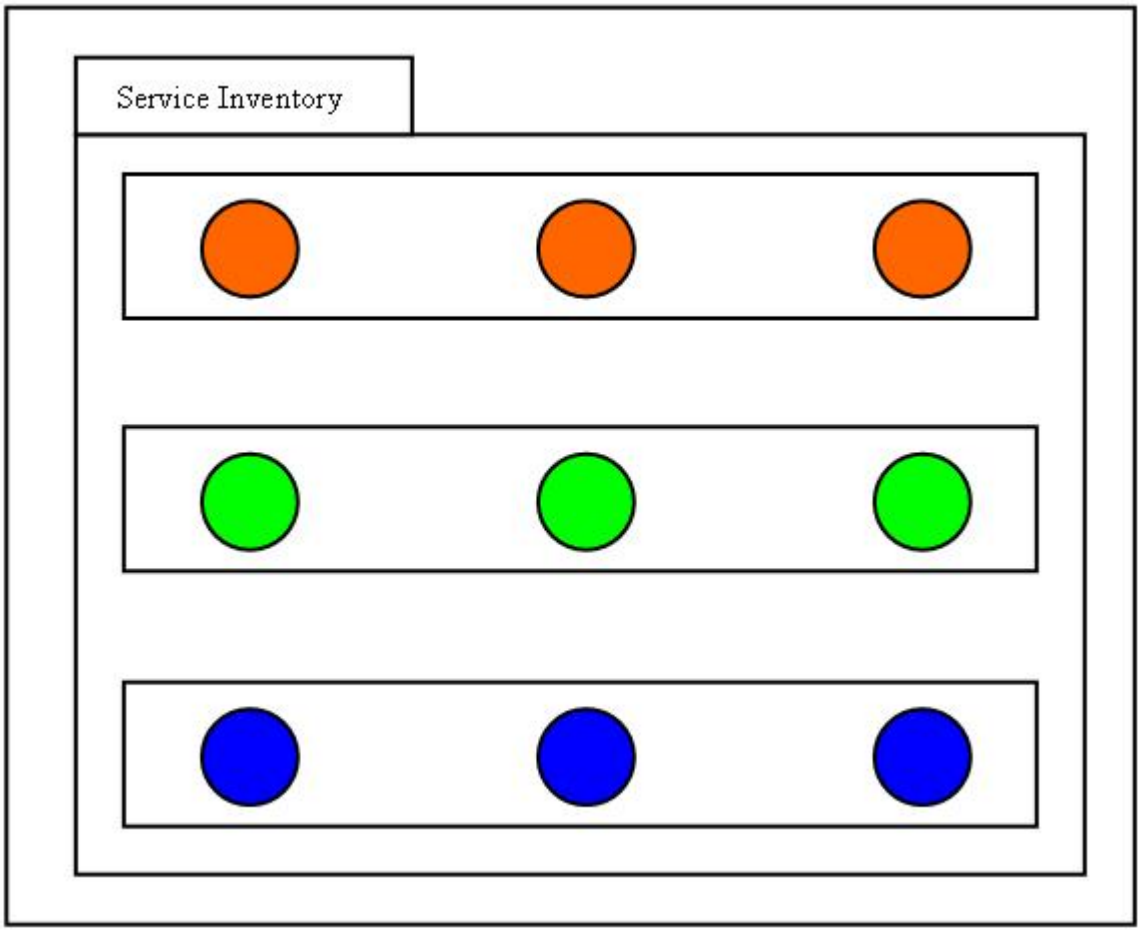


Diagram B
A service inventory divided into layers where each layer contains the same type of logic.

Applying this pattern requires creating a service inventory blueprint, a list of services with associated functionality. Next, group the services into layers according to function. Adopting a common layering strategy across the enterprise facilitates reuse in other applications, because developers don't have as much to learn (or invent) when they join a project. One common layering uses task, entity and utility layers..

An alternative layering from Bieberstein et al., involves five layers, namely Enterprise, Process, Service, Component and Object.

The Service Layers pattern invokes a specific service architecture.

The top-down service delivery approach facilitates the use of this pattern.

Chapter 14

Service Loose Coupling & Service Discoverability Principle

Service Loose Coupling

Within the service-orientation design paradigm, **Service Loose Coupling** is a design principle that is applied to the services in order to ensure that the service contract is not tightly coupled to the service consumers and to the underlying service logic and implementation. This results in service contracts that could be freely evolved without affecting either the service consumers or the service implementation.

Purpose

The concept of loose coupling within SOA is directly influenced by the object-oriented design paradigm, whereby the objective is to reduce coupling between classes in order to foster an environment where both the classes, although somehow related to each other, can be changed in a manner that such a change does not break the existing relationship, which is necessary for the working of a software program. The same concept applies within SOA world as well, however, within SOA particular emphasis is on the service contract as the service contract acts as an interface through which service consumers communicate with the service logic and vice-versa. Apart from this, SOA strongly advocates development of physically independent service contracts from the service logic (Decoupled Contract design pattern) in favor of interoperability and technology independence. As the contracts are physically independent, there is a need to not only look into the coupling between service consumers and service contracts but also between service contracts and their underlying logic and implementation. This is where the application of this design principle helps in identifying the various types of couplings that exist (inter service as well as intra service) and how to design the contracts in order to minimize negative coupling types and maximize positive coupling types. A service-oriented solution that consists of services having loosely coupled contracts directly supports the increased vendor diversity options and increased interoperability goals of service-orientation.

Application

The application of the Service Loose Coupling design principle requires delving into the different types of couplings that exist between the service consumer and the service contract as well as the service contract and the service's implementation. Only by understanding these different types, their impact on the service-orientation can be correctly analyzed.

Coupling Types

Logic-to-Contract

The Service Loose Coupling design principle dictates that this kind of coupling should to be favored so that the service logic is developed exclusively in support of the service contract. However, this requires following the 'contract first' approach as advocated by the Standardized Service Contract principle so that the service logic is coupled to a standardized contract. This way the service contract is not coupled to the logic so the logic could be replaced in future if need be without affecting the service consumers.

Contract-to-Logic

This type of coupling exists when the contract is built based on existing logic e.g. through automated tools. This is a negative form of coupling and needs to be avoided as it inhibits the evolution of service contract. This is because the service contract is not designed independently according to the design standards and is dictated by the underlying logic.

Contract-to-Implementation

When contracts are designed in a manner that they are based on the underlying implementation details e.g. data models used within the underlying database, this results in a negative form of coupling that needs to be avoided. This way, a change in the underlying implementation will require a corresponding change in the service contract. This type of coupling can be reduced with the introduction of a façade component in between the service logic and its implementation as advocated by the Service Façade design pattern.

Contract-to-Technology

A contract that exposes proprietary technology elements used by the service logic e.g. a contract based on .NET Remoting technology, forms a negative form of coupling as the service consumers are limited to that particular technology. This greatly hampers the service's ability to be counted as an interoperable enterprise-resource.

Contract-to-Functional

This type of coupling normally exists when the service contract is developed by keeping a particular kind of consumer in mind e.g. services built to enable communication with a business partner or a service that executes a part of the business process logic or is itself the parent controller service in a service composition that executes the business process logic. This is also a negative form of coupling and needs to be avoided. Although in case of agnostic services there is a clear need to reduce this coupling type, however, in case of non-agnostic services e.g. the task services, existence of such coupling is intentional because the service is not required to be particularly reusable and hence could be tightly coupled to a particular consumer for better efficiency.

Consumer-to-Implementation

This is a negative form of coupling that exists because the service consumers access the service directly either via its logic or implementation. This can happen because of number of reasons. For example, the service consumers used to access the current service through streamlined proprietary interfaces long before it actually existed as a service i.e. before the move towards service-orientation. The application of the Contract Centralization design pattern helps to avoid this kind of coupling.

Consumer-to-Contract

This is a favorable type of coupling as it helps to evolve the service without impacting its consumers. However, it is quite important to bear in mind that this coupling should only be restricted to the service contract and should not leak into the service architecture. This could happen if all of the negative contract related coupling types are not addressed, consequently the service consumer can easily become coupled to the service implementation, logic or technology.

Considerations

Designing service contracts that are totally decoupled from their internal and external surroundings would no doubt result in services that are interoperable and scalable but on the other side, this may create contracts that have capabilities which are too generic or the capabilities' message exchange is too generic which would result in more roundtrips, requiring increased processing resources and time.

Analyzing all of the above different types of coupling requires extra time and efforts and may increase the delivery time of services. Consequently, there is a need to apply this design principle to a meaningful extent as set by the design standards within the individual organization.

Service Discoverability Principle

Service Discoverability is a design principle, applied within the service-orientation design paradigm, which emphasizes making services discoverable by adding interpretable meta-data to increase service reuse and decrease the chance of developing services that overlap in function. By making services easily discoverable, this design principle indirectly makes services more interoperable.

Purpose

The use of any piece of information directly relates to how discoverable it is. Extending the same concept to software development, the reuse potential of a software program cannot be achieved if no one knows even if it exists or not. Secondly, this information is only good if someone can properly understand it, i.e., it depends upon the quality of the meta-information as well to make the best use of the software program.

In case of a service-oriented solution, because of the emphasis placed on service reusability, it stands very clear that opportunities should exist for their reuse, which is only possible if they are discoverable in the first place. To make services discoverable, following set of activities need to be performed:

1. Document the information about the service in a consistent manner.
2. Store the documented information in a searchable repository.
3. Enable others to search for the documented information in an efficient manner.

Apart from increasing the reuse potential of the services, discoverability is also required to avoid development of solution logic that is already contained in an existing service. To design services that are not only discoverable but also provide interpretable information about their capabilities, the Service Discoverability principle provides guidelines that could be applied during the service-oriented analysis phase of the service delivery process.

Application

The application of this principle requires collecting information about the service during the service analysis phase as during this phase; maximum information is available about the service's functional context and the capabilities of the service. At this stage, the domain knowledge of the business experts could also be enlisted to document meta-data about the service. In the service-oriented design phase, the already gathered meta-data could be made part of the service contract. The OASIS SOA-RM standard specifies *service description* as an artifact that represents service meta-data.

To make the service meta-data accessible to interested parties, it must be centrally accessible. This could either be done by publishing the service-meta to a dedicated 'service registry' or by simply placing this information in a 'shared directory'. In case of a

'service registry', the repository can also be used to include QoS, SLA and the current state of a service.

Meta-data Types

Functional

This is the basic type of meta-information that expresses the functional context of the service and the details about the service's capabilities. The application of the Standardized Service Contract principle helps to create the basic functional meta-data in a consistent manner. The same standardization should be applied when the same meta-information is being outside the technical contract of the service e.g. when publishing information to a service registry.

Quality of Service

To know about the service behavior and its limitations, all of this information needs to be documented within the service registry so that the potential consumers can use this meta-information by comparing it against their performance requirements.

Considerations

The effective application of this design principle requires that the meta-information recorded against each service needs to be consistent and meaningful. This is only possible if organization-wide standards exist that enforce service developers to record the required meta-data in a consistent way. The information recorded as the meta-data for the service needs to be presented in a way so that both technical and non-technical IT experts can understand the purpose and the capabilities of the service, as an evaluation of the service may be required by the business people before the service is authorized to be used.

This principle is best applied during the service-oriented analysis phase as during this time, all the details about the service's purpose and functionality are available.

Although most of the service design principles support each other in a positive manner, however, in case of Service Abstraction and Service discoverability principle, there exists an inversely proportional relationship. This is because as more and more details about the service are hidden away from the service consumers, less discoverable information is available for discovering the service. This could be addressed by carefully recording the service meta-information so that the inner workings of the service are not documented within this meta-information.

Chapter 15

Logic Centralization Pattern & Service Composability Principle

Logic Centralization Pattern

Logic Centralization is a design pattern, applied within the service-orientation design paradigm, whose application aims to increase the reusability potential of agnostic logic by ensuring that services do not contain redundant agnostic logic and that any reusable logic should only be represented by a service that has the most suitable functional context.

Rationale

As more and more services are developed, there is a constant risk that services with redundant functionality may be created. Although the application of the Service Normalization design pattern does help to eliminate this redundancy, however, just by having a set of normalized services on its own, does not guarantee that they would be reused as originally envisaged. In case of agnostic services, this issue can severely restrict the actual reuse of such services because a project team (Team A) may decide no to reuse an existing service, e.g. it requires data that corresponds to a complex schema, and instead develop a lightweight service that just does the job. As a result, the same reusable logic now exists with two different services, whereas the existing service should have been evolved even if it did not contain the most suitable flavor of the functionality. This effect gets multiplied when another team (Team B) hoping to find the functionality within the existing service, as the boundary of the service does cover the required functionality, fails to find it and instead start using the newly created service by Team A. Consequently, the actual reusability of the original agnostic service drops and at the same time creates governance problem as far as the maintenance of the original and new services is concerned because now reusable logic exists in a decentralized manner.
In order to ensure that a particular type of reusable solution logic is only enclosed by one specific agnostic service, the Logic Centralization design pattern dictates that design standards need to be established that force the proper use of agnostic services. This gives the service consumers the confidence that they are accessing the functionality through the correct service.

Usage

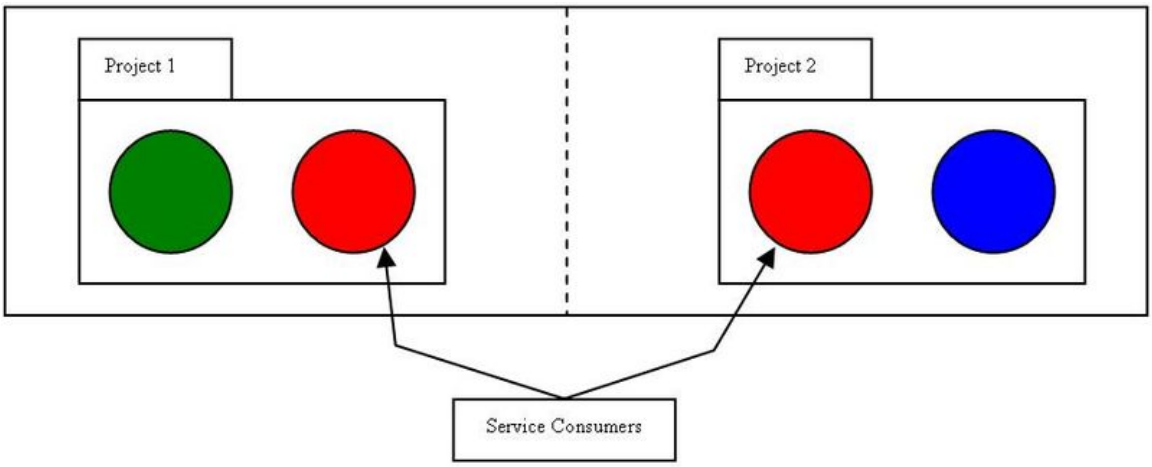


Diagram A
Instead of using existing red service, Project Team 1 resorts to creating a new redundant red service as it was easy to develop a new service that was more streamlined with their short-term requirements.

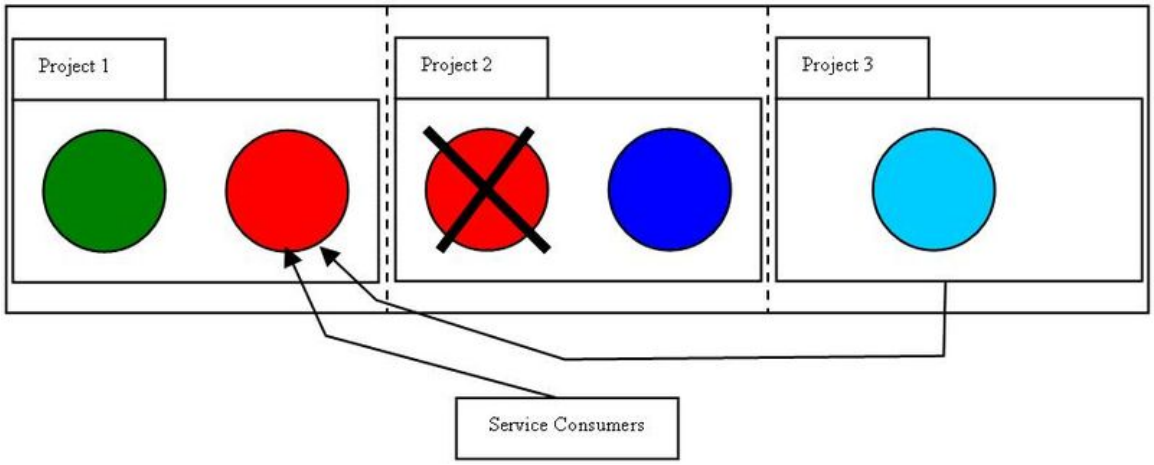


Diagram A
In the presence of an enterprise-wide design standard, service consumers' access to the redundant red service created by Project Team 2 is prohibited and instead they are forced to use the existing red service created by Project Team 1. Similarly, Project Team 3 is prohibited to create any new service whose functionality falls within the existing red service, as a result Project Team 3 uses/evolves the existing red service created by Project Team 1.

The application of this design pattern requires setting up 'official endpoints' (services) that represent a particular type of functionality i.e. the functionality that falls under a particular type of functional domain. Access to any other services, which may offer the same functionality, is prohibited and only one service is made accessible for a particular type of functionality. By restricting access to other services, the governance burden is

reduced as now the logic is confined within a single service. Whenever a new functionality is required, which is not currently offered by any of the existing services, then first the functional contexts of the existing services need to be checked and if the new functionality falls under the boundary of an already existing service, then it should be added to that service. This requires an enterprise wide standard that enforces the centralization of logic. In order to make sure that service developers are aware of the service boundaries, Metadata Centralization design pattern could be applied. This helps in creating a centralized repository of information about the functional contexts and the functionality provided by the services. The Logic Centralization design pattern when applied together with the Contract Centralization design pattern, constitute the Official Endpoint design pattern. The application of the Logic Centralization design pattern further helps in the application of the Service Reusability and the Service Composability design principles by ensuring that each service contains the right type of reusable functionality so that it can be composed repeatedly.

Considerations

In order to apply this design pattern, it needs to be ensured that all the project teams across the enterprise understand and agree to the proper use of agnostic services and do not create any new services that only serve the short-term requirements of a project. This can also impact the project delivery times as the use of existing agnostic services (and to evolve them as per the guidelines of this pattern) would require increased time and efforts. This is because the services in the current project may not be able to make use of the agnostic services until their own design is changed.

Service Composability Principle

Service Composability is a design principle, applied within the service-orientation design paradigm, which encourages the design of services in a manner so that they can be reused in multiple solutions that are themselves made up of composed services. The ability of the service to be recomposed should be independent of the size and complexity of the service composition.
This principle is directly related to the agility promised by SOA as it promotes composing new solutions by reusing existing services.

Purpose

The concept of developing software out of independently existing components encourages the concept of composition. This is the underlying concept within object-orientation where the end product is composed of several interlinked objects that have the ability to become part of multiple software solutions, no matter how complex the solution is. The same composition concept is inherited by service-orientation, whereby a business

process is automated by combining multiple services. However, within service-orientation there is even greater focus on building services that can be composed and recomposed within multiple solutions in order to provide the agility promised by the SOA. As a result of this emphasis, some guidelines are required in order to develop services that can be effectively aggregated into multiple solutions.
The Service Composability principle provides design considerations that help towards designing composable services with a view to encourage service reuse as much as possible. The guidelines provides by this principle prepare the service so that it is ready to participate in service compositions without requiring any further design changes.

Application

The application of this design principle requires designing services in a manner so that they can be used in a service composition either as a service that controls other services i.e. a controller service, or as a service that provides functionality to other services in the composition without further composing other services i.e. a composition member.
In order for the service to provide this dual functionality, the service contract needs to be designed in a manner so that it presents functionality based on varying levels of input and output data. In case if it is required to participate as a composition member, then usually the input parameters to the service would be more fine grained as compared to the situation when it is required to participate as a composition controller. A heavily reused service needs to be as stateless as possible (Service Statelessness principle) so that it can provide optimum performance when composed within multiple service compositions.
The effectiveness of this principle depends upon the extent to which rest of the design principles have been applied successfully. The application of the Standardized Service Contract principle enables the services to be interoperable with other and helps to keep the composition design rather simpler by avoiding the need to perform runtime data model transformation. By applying the Service Loose Coupling principle, a service could be recomposed with the confidence that it would not create any form of negative coupling with the other service in the composition. The application of the Service Autonomy and the Service Statelessness principles increase the reliability and availability of the service so that it be reused in multiple service compositions with increased confidence.

Considerations

In order for the service to be an efficient service controller as well as a service member, the underlying technology architecture needs to provide a runtime environment that is scalable and can support the statelessness required by the service. Similarly as the service compositions increase in size, the storage and retrieval of the context data, related to the runtime interaction of the services, may need to be delegated to the runtime environment instead of the services managing this context data in order to make the service composition more efficient.
As more and more service compositions are built, there is a tendency of getting dependent on a service that is highly reused. This requires careful analysis during the design of the service compositions and considering alternate standby services for critical functionality. On the other hand, it may become difficult to evolve a service that is now

become a part of multiple service compositions. This could be addressed by the application of the Concurrent Contracts design pattern that advocates maintaining multiple concurrent contracts for a service. This way the service can evolve while providing backward compatibility.
Some of the factors that determine the composability potential of a service include:

- Ability to provide functionality at different levels within a business process.
- Message Exchange pattern
- Whether the service supports transactions and rollback/compensation features.
- Support for exception handling.
- The availability of meta-data about service capabilities and behavior.

Chapter 16

Utility Abstraction Pattern & Standardized Service Contract

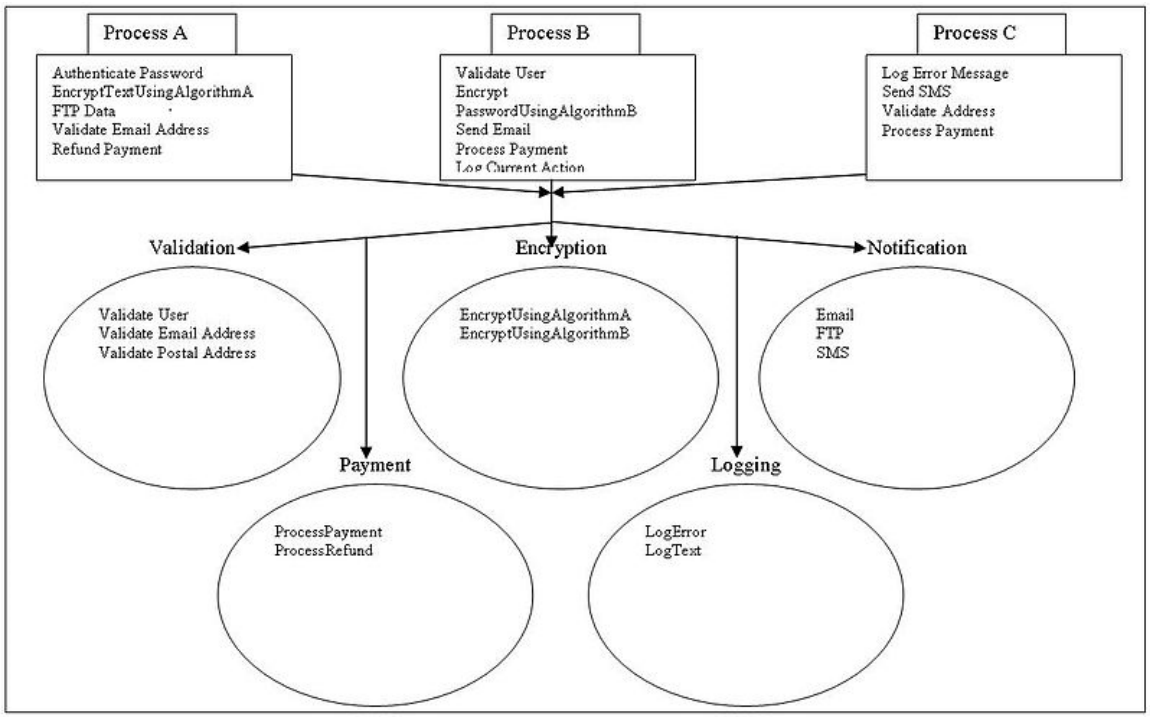
Utility Abstraction Pattern

Utility Abstraction is a design pattern, applied within the service-orientation design paradigm, which advocates designing services that provide cross-cutting non-business related functionality, which can be positioned as utility resources to automate multiple business processes.

Rationale

In order to automate a business process, the required solution logic is based on the corresponding business work-flow and the business rules. Although the business specific logic can be derived from the business domain, however, in order to complete the automation of the business process, some general purpose logic is normally required that cannot be derived from the business domain. Such logic usually offers low level generic functionality that is commonly required to automate multiple business processes e.g. logging incoming service request messages for metering purposes. As different business processes are analyzed, the corresponding general purpose utility logic is identified and delivered as part of the main business processing solution logic (within the same service). Although this method works fine for fulfilling the short-term project requirements, it gets difficult to keep different services, each containing some level of utility logic, in-sync in the long run when it comes to making a change to the contained utility functionality e.g. due to an upgrade of the underlying technology resources (databases, legacy systems). On the other hand, it not only denormalizes the service inventory, due to the redundancy of the utility logic, but also loses the opportunity of making such general purpose logic available for reuse to automate multiple business processes.
In order to address the above issues, the Utility Abstraction design pattern dictates the separation of generic processing logic from the business process-specific logic into a separate group of services known as the utility services.

Usage



Abstracting Utility Functions
Processes A, B and C all contain utility functions which can be of interest to other services. As a result, five different utility functional contexts are created and the related functionality from each process is added to the most relevant functional context. Doing so also eliminates any redundant functionality.

This design pattern provides practical guidelines for designing the utility service layer as advocated by the Service Layers design pattern. These utility services are designed by analyzing the common processing requirements of business services (services that contain business process-specific logic) and then developing the required functionality represented by a meaningful functional context. The functional contexts of utility services would tend to be more technology oriented as compared to entity services as utility services mostly provide technology interfacing functionality e.g. a wrapper service that talks to a legacy database or a service that provides data conversion functionality between different formats.
It is important to understand that the functional contexts identified for the utility services need to correctly represent the functionality contained within utility services. This places an increased burden on correctly identifying such functional contexts which may prove difficult as the contained functionality is not directly linked to a particular business context and hence cannot be easily categorized into meaningful functional contexts. Some of the suggestions include:

- Setting the functional boundary according to the specific technology to which the utility service provides an interface e.g. a utility service for talking to a ERP system.
- Identifying the functional boundary according to the category of utility functions e.g. a utility service that performs encryption and decryption of messages.

The utility services are mostly used as helper services from multiple business process-specific logic, each having specific requirements, consequently it is better to design utility services that are based on targeted and specific functional contexts rather than vague functional contexts that pack a range of functions. This is important as it not only helps to keep the maintenance overhead of such services to a minimum but also increases the reuse potential of utility functionality because the service consumers can easily identify the required generic functionality as it is not bundled up in a broad functional context that does not correctly represent all of its contained functionality. The application of Canonical Expression design pattern can further enhance the effectiveness of such targeted functional contexts by advocating the use of standardized naming conventions that correctly describe these functional contexts.

Considerations

By moving the utility logic into separate services, inter-service communication is increased that can introduce latency as well as increased processing overhead. Similarly, due to the distribution of generic logic, the overall design of the service composition will become complex and difficult to maintain.

Standardized Service Contract

The **Standardized Service Contract** is a design principle, applied within the service-orientation design paradigm, in order to guarantee that the service contracts within a service inventory (enterprise or domain) adhere to the same set of design standards, which results in standardized service contracts across the service inventory.

Purpose

The agility promised by an SOA is usually measured in terms of the reusability level of its contained services. However, this reusability is directly related to the way service capabilities have been defined within the service contract, as a service that is built upon a potentially reusable functional context but whose contract does not convey this reusability in an appropriate manner is not going to achieve its reusability potential. Within service-oriented solutions, a service contract represents a fundamental artifact as this is the only medium through which the services interact with each other or with other potential consumer programs. Therefore, there exists a strong need to standardize the

service contracts in order to make services reusable and recomposable as much as possible. In order to achieve this, the Standardized Service Contract design principle needs to be applied as its application results in standardized service contracts that are based on design standards as set within a service inventory.

One of its goals is to reduce the need for data transformations as two services interact with each other, which can be achieved if the service contracts use standardized data models e.g. XML schemas if the services have been implemented as web services. This also helps in making services more interoperable. Another important goal of this design pattern is to use a standardized way of expressing service capabilities so that their purpose and ability can be easily understood at design time.

Application

A technical service contract is usually composed of a WSDL document, XML schema(s) and policy document(s). Consequently, this principle needs to be applied across three areas of a service contract as described below:

Functional Expression Standardization

The service's operations need to be defined using standardized naming conventions. This would also apply to the constituent input and out message names and their corresponding type names. This helps to increase the service contract's correct interpretation, which in turn increases service’s reuse and interoperability. When service contracts clearly express their capabilities, the chance of service duplication is also reduced.

Data Model Standardization

Two services exchanging messages based on the same type of data e.g. a purchase order, might model that data according to different schemas, which requires data model transformation. This is clearly an overhead and stands in the way of service interoperability and reuse. In order to avoid this transformation, the Standardized Service Contract principle requires developing standardized data models, which further helps in the creation of a standardized data representation architecture that could be reused across the enterprise for defining standardized service capabilities. The objectives of data model standardization are directly supported by the Schema Centralization design pattern, which further helps in the creation of centrally governed schemas.

Policy Standardization

Service policies represent the terms of usage for a service. So in order for a service to be reusable, its behavioral requirements need to be expressed in a consistent manner using standardized policy expressions that are based on industry standard vocabularies. This type of standardization further promotes separation of policies from the service contracts into individual policy documents in order to allow centralized governance. In some cases,

two policies, although syntactically different, might mean the same thing, therefore, design standards need to be in place that dictate the acceptable policy structure.

Considerations

The application of this design principle depends upon the existence of design standards on the service inventory level. This would require additional resources in terms of time and effort. Secondly, in order to apply this design principle effectively, the actual contract needs to be physically isolated from the service logic and implementation so that it could be based upon industry standards. This could be achieved by the application of the Decoupled Contract design pattern. Also that the ‘contract first’ approach needs to be followed so that the underlying logic only makes use of standardized data models. Furthermore, the requirement for centralized data models may end up in the transmission of redundant data between services as the actual data required by a service may only be a subset of the data as expressed by the standardized schema imposed on the service.

Chapter 17

Service-Oriented Programming

Service-oriented programming (SOP) is a programming paradigm that uses "services" as the unit of computer work, to design and implement integrated business applications and mission critical software programs. Services can represent steps of business processes and thus one of the main applications of this paradigm is the cost-effective delivery of standalone or composite business applications that can “integrate from the inside-out”

Introduction

SOP inherently promotes Service Oriented Architecture (SOA), however, it is not the same as SOA. While SOA focuses on communication between systems using “services”, SOP provides a new technique to build agile application modules using in-memory services as the unit of work.

An in-memory service in SOP can be transparently externalized as a Web Service operation. Due to language and platform independent Web Service standards, SOP embraces all existing programming paradigms, languages and platforms. In SOP, the design of the programs pivot around the semantics of service calls, logical routing and data flow description across well-defined service interfaces. All SOP program modules are encapsulated as services and a service can be composed of other nested services in a hierarchical manner with virtually limitless depth to this service stack hierarchy. A composite service can also contain programming constructs some of which are specific and unique to SOP. A service can be an externalized component from another system accessed either through using Web Service standards or any proprietary API through an in-memory plug-in mechanism.

While SOP supports the basic programming constructs for sequencing, selection and iteration, it is differentiated with a slew of new programming constructs that provide built-in native ability geared towards data list manipulation, data integration, automated multithreading of service modules, declarative context management and synchronization of services. SOP design enables programmers to semantically synchronize the execution of services in order to guarantee that it is correct, or to declare a service module as a transaction boundary with automated commit/rollback behavior.

Semantic design tools and runtime automation platforms can be built to support the fundamental concepts of SOP. For example, a Service Virtual Machine (SVM) that automatically creates service objects as units of work and manages their context can be designed to run based on the SOP program metadata stored in XML and created by a design-time automation tool. In SOA terms, the SVM is both a service producer and a service consumer.

Fundamental Concepts

SOP concepts provide a robust base for a semantic approach to programming integration and application logic. There are three significant benefits to this approach:

- Semantically, it can raise the level of abstraction for creating composite business applications and thus significantly increase responsiveness to change (i.e. business agility)
- Gives rise to the unification of integration and software component development techniques under a single concept and thus significantly reduces the complexity of integration. This unified approach enables "inside-out integration" without the need to replicate data, therefore, significantly reducing the cost and complexity of the overall solution
- Automate multi-threading and virtualization of applications at the granular (unit-of-work) level.

The following are some of the key concepts of SOP:

Encapsulation

In SOP, in-memory software modules are strictly encapsulated through well-defined service interfaces that can be externalized on-demand as Web Service operations. This minimal unit of encapsulation maximizes the opportunities for reusability within other in-memory service modules as well as across existing and legacy software assets. By using service interfaces for information hiding, SOP extends the service-oriented design principles used in SOA to achieve separation of concerns across in-memory service modules.

Service Interface

A service interface in SOP is an in-memory object that describes a well-defined software task with well-defined input and output data structures. Service interfaces can be grouped into packages. An SOP service interface can be externalized as a WSDL operation and a single service or a package of services can be described using WSDL. Furthermore, service interfaces can be assigned to one or many service groups based on shared properties.

In SOP, runtime properties stored on the service interface metadata serve as a contract with the Service Virtual Machine (SVM). One example for the use of runtime properties

is that in declarative service synchronization. A service interface can be declared as a fully synchronized interface, meaning that only a single instance of that service can run at any given time. Or, it can be synchronized based on the actual value of key inputs at runtime, meaning that no two service instances of that service with the same value for their key input data can run at the same time. Furthermore, synchronization can be declared across services interfaces that belong to the same service group. For example, if two services, 'CreditAccount' and 'DebitAccount', belong to the same synchronization service group and are synchronized on the accountName input field, then no two instances of 'CreditAccount' and 'DebitAccount' with the same account name can execute at the same time.

Service Invoker

A service invoker makes service requests. It is a pluggable in-memory interface that abstracts the location of a service producer as well as the communication protocol, used between the consumer and producer when going across computer memory, from the SOP runtime environment such as an SVM. The producer can be in-process (i.e. in-memory), outside the process on the same server machine, or virtualized across a set of networked server machines. The use of a service invoker in SOP is the key to location transparency and virtualization. Another significant feature of the service invoker layer is the ability to optimize bandwidth and throughput when communicating across machines. For example, a "SOAP Invoker" is the default service invoker for remote communication across machines using the Web Service standards. This invoker can be dynamically swapped out if, for example, the producer and consumer wish to communicate through a packed proprietary API for better security and more efficient use of bandwidth.

Service Listener

A service listener receives service requests. It is a pluggable in-memory interface that abstracts the communication protocol for incoming service requests made to the SOP runtime environment such as the SVM. Through this abstract layer, the SOP runtime environment can be virtually embedded within the memory address of any traditional programming environment or application service.

Service Implementation

In SOP, a service module can be either implemented as a Composite or Atomic service. It is important to note that Service modules built through the SOP paradigm have an extroverted nature and can be transparently externalized through standards such as SOAP or any proprietary protocol.

Semantic-based Approach

One of the most important characteristic of SOP is that it can support a fully semantic-based approach to programming. Furthermore, this semantic-based approach can be layered into a visual environment built on top of a fully metadata-driven layer for storing

the service interface and service module definitions. Furthermore, if the SOP runtime is supported by a SVM capable of interpreting the metadata layer, the need for automatic code generation can be eliminated. The result is tremendous productivity gain during development, ease of testing and significant agility in deployment.

Service Implementation: Composite Service

A composite service implementation is the semantic definition of a service module based on SOP techniques and concepts. If you look inside of a black-boxed interface definition of a composite service, you may see other service interfaces connected to each other and connected to SOP programming constructs. A Composite service has a recursive definition meaning that any service inside ("inner service") may be another atomic or composite service. An inner service may be a recursive reference to the same containing composite service.

Programming Constructs

SOP supports the basic programming constructs for sequencing, selection and iteration as well as built-in, advance behavior. Furthermore, SOP supports semantic constructs for automatic data mapping, translation, manipulation and flow across inner services of a composite service.

Sequencing

A service inside of the definition of a composite service (an "inner service") is implicitly sequenced through the semantic connectivity of built-in success or failure ports of other inner services with its built-in activation port. When an inner service runs successfully, all the inner services connected to its success port will run next. If an inner service fails, all the services connected to its failure port will run next.

Selection

Logical selection is accomplished through data-driven branching constructs and other configurable constructs. In general, configurable constructs are services built into the SOP platform with inputs and outputs that can assume the input/output shape of other connected services. For example, a configurable construct used for filtering output data of services can take a list of Sales orders, Purchase orders or any other data structure, and filter its data based on user declared filter properties stored on the interface of that instance of the filter construct. In this example, the structure to be filtered becomes the input of the particular instance of the filter construct and the same structure representing the filtered data becomes the output of the configurable construct.

Iteration

A composite service can be declared to loop. The loop can be bound by a fixed number of iterations with an optional built-in delay between iterations and it can dynamically

terminate using a “service exit with success” or “service exit with failure” construct inside of the looping composite service. Furthermore, any service interface can automatically run in a loop or “foreach” mode, if it is supplied with two or more input components upon automatic preparation. This behavior is supported at design-time when a data list structure from one service is connected to a service that takes a single data structure (i.e. non-plural) as its input. If a runtime property of the composite service interface is declared to support “foreach” in parallel, then the runtime automation environment can automatically multi-thread the loop and run it in parallel. This is an example of how SOP programming constructs provide built-in advanced functionality.

Data Transformation, Mapping, and Translation

Data mapping, translation, and transformation constructs enable automatic transfer of data across inner services. An inner-service is prepared to run, when it is activated and all of its input dependencies are resolved. All the prepared inner-services within a composite service run in a parallel burst called a “Hypercycle”. This is one of the means by which automatic parallel-processing is supported in SOP. The definition of a composite service contains an implicit directed graph of inner service dependencies. The runtime environment for SOP can create an execution graph based on this directed graph by automatically instantiating and running inner services in parallel whenever possible.

Exception Handling

Exception handling is an run time error in java. Exception handling in SOP is simply accomplished by connecting the failure port of inner services to another inner service, or to a programming construct. ‘Exit with Failure’ and “Exit with Success’ constructs are examples of constructs used for exception handling. If no action is taken on the failure port of a service, then the outer (parent) service will automatically fail and the standard output messages from the failed inner service will automatically bubble up to the standard output of the parent.

Transactional Boundary

A composite service can be declared as a transaction boundary. The runtime environment for SOP automatically creates and manages a hierarchical context for composite service objects which are used as a transaction boundary. This context automatically commits or rollbacks upon the successful execution of the composite service.

Service Compensation

Special composite services, called compensation services, can be associated with any service within SOP. When a composite service that is declared as a transaction boundary fails without an exception handling routing, the SOP runtime environment automatically dispatches the compensation services associated with all the inner services which have already executed successfully.

Service Implementation: Atomic Service

An Atomic service is an in-memory extension of the SOP runtime environment through a Service Native Interface (SNI) it is essentially a plug-in mechanism. For example, if SOP is automated through an SVM, a service plug-in is dynamically loaded into the SVM when any associated service is consumed. An example of a service plug-in would be a SOAP communicator plug-in that can on-the-fly translate any in-memory service input data to a Web Service SOAP request, post it to a service producer, and then translate the corresponding SOAP response to in-memory output data on the service. Another example of a service plug-in is a standard database SQL plug-in that supports data access, modification and query operations. A further example that can help establish the fundamental importance of atomic services and service plug-ins is using a service invoker as a service plug-in to transparently virtualize services across different instances of an SOP platform. This unique, component-level virtualization is termed "Service Grid Virtualization" in order to distinguish it from traditional application, or process-level virtualization.

Cross-cutting Concepts

SOP presents significant opportunities to support cross-cutting concerns for all applications built using the SOP technique. The following sections define some of these opportunities:

Service Instrumentation

The SOP runtime environment can systematically provide built-in and optimized profiling, logging and metering for all services in real-time.

Declarative & Context-sensitive Service Caching

Based on declared key input values of a service instance, the outputs of a non time-sensitive inner service can be cached by the SOP runtime environment when running in the context of a particular composite service. When a service is cached for particular key input values, the SOP runtime environment fetches the cached outputs corresponding to the keyed inputs from its service cache instead of consuming the service. Availability of this built-in mechanism to the SOP application developer can significantly reduce the load on back-end systems.

Service Triggers

SOP provides a mechanism for associating a special kind of composite service, trigger service, to any other service. When that service is consumed, the SOP platform automatically creates and consumes an instance of the associated trigger service with an in-memory copy of the inputs of the triggering service. This consumption is non-intrusive to the execution of the triggering service. A service trigger can be declared to run upon activation, failure or success completion of the triggering service.

Inter-Service Communication

In addition to the ability to call any service, Service Request Events and Shared Memory are two of the SOP built-in mechanisms provided for inter-service communication. The consumption of a service is treated as an Event in SOP. SOP provides a correlation-based event mechanism that results in the pre-emption of a running composite that has declared, through a “wait” construct, the need to wait for one or more other service consumption events to happen with specified input data values. The execution of the composite service continues when services are consumed with specific correlation key inputs associated with the wait construct. SOP also provides a shared memory space with access control where services can access and update a well-defined [[data structure that is similar to the input/output structure of services. The shared memory mechanism within SOP can be programmatically accessed through service interfaces.

Service Overrides

In SOP, customizations are managed through an inventive feature called Service Overrides. Through this feature, a service implementation can be statically or dynamically overridden by one of many possible implementations at runtime. This feature is analogous to polymorphism in object-oriented programming. Each possible override implementation can be associated to one or more override configuration portfolios in order to manage activation of groups of related overrides throughout different SOP application installations at the time of deployment.

Consumer Account Provisioning

Select services can be deployed securely for external programmatic consumption by a presentation (GUI) layer, or other applications. Once service accounts are defined, the SOP runtime environment automatically manages access through consumer account provisioning mechanisms.

Security

The SOP runtime environment can systematically provide built-in authentication and service authorization. For the purpose of authorization, SOP development projects, consumer accounts, packages and services are treated as resources with access control. In this way, the SOP runtime environment can provide built-in authorization. Standards or proprietary authorization and communication security is customized through service overrides, plug-in invoker and service listener modules.

Virtualization and Automatic Multithreading

Since all artifacts of SOP are well encapsulated services and all SOP mechanisms, such as shared memory, can be provided as distributable services, large scale virtualization can be automated by the SOP runtime environment. Also, the hierarchical service stack of a composite service with the multiple execution graphs associated to its inner services, at

each level, provides tremendous opportunities for automated multi-threading to the SOP runtime environment.

History

The term ‘Service Oriented Programming’ was first published in 2002 by Alberto Sillitti, Tullio Vernazza and Giancarlo Succi in a book called "Software Reuse: Methods, Techniques, and Tools.” SOP, as described above, is neither influenced by nor reflects Sillitti, Vernazza and Succi’s use of the term.

Today, the SOP paradigm is in the early stages of mainstream adoption. There are four market drivers fueling this adoption:

- Multi-core Processor Architecture: due to heat dissipation issues with increasing processor clock speeds beyond 4 GHZ, the leading processor vendors such as Intel have turned to multi-core architecture to deliver ever increasing performance. Refer to “The Free Lunch Is Over" This change in processor architecture forces a change in the way we develop our software modules and applications: applications must be written for concurrency in order to utilize multi-core processors and writing concurrent programs is a challenging task. SOP provides a built-in opportunity for automated multithreading.
- Application Virtualization: SOP promotes built-in micro control over location transparency of the service constituents of any service module. This results in automatic and granular virtualization of application components (versus an entire application process) across a cluster or grid of SOP runtime platforms.
- Service-oriented architecture (SOA) and demand for integrated and composite applications: in the beginning, the adoption of SOP will follow the adoption curve of SOA with a small lag. This is because services generated through SOA can be easily assembled and consumed through SOP. The more Web services proliferate, the more it makes sense to take advantage of the semantic nature of SOP. On the other hand, since SOA is inherent in SOP, SOP provides a cost-effective way to deliver SOA to mainstream markets.
- Software as a Service (SaaS): capabilities of the current SaaS platforms cannot address the customization and integration complexities required by large enterprises. SOP can significantly reduce the complexity of integration and customization. This will drive SOP into the next generation SaaS platforms.