Lecture 4

Polymorphism and Binding
Composite, Iterator, Visitor, Strategy, State, Bridge, Abstract Factory

Polymorphism

The Dynamic Model:

Behavioral Aspects

"Well," says Buck, "a feud is this way. A man has a quarrel with another man, and kills him; then that other man's brother kills *him*; then the other brothers, on both sides, goes for one another; then the *cousins* chip in—and by and by everybody's killed off, and there ain't no more feud. But it's kind of slow, and takes a long time. . . . ."—*Adventures of Huckleberry Finn*

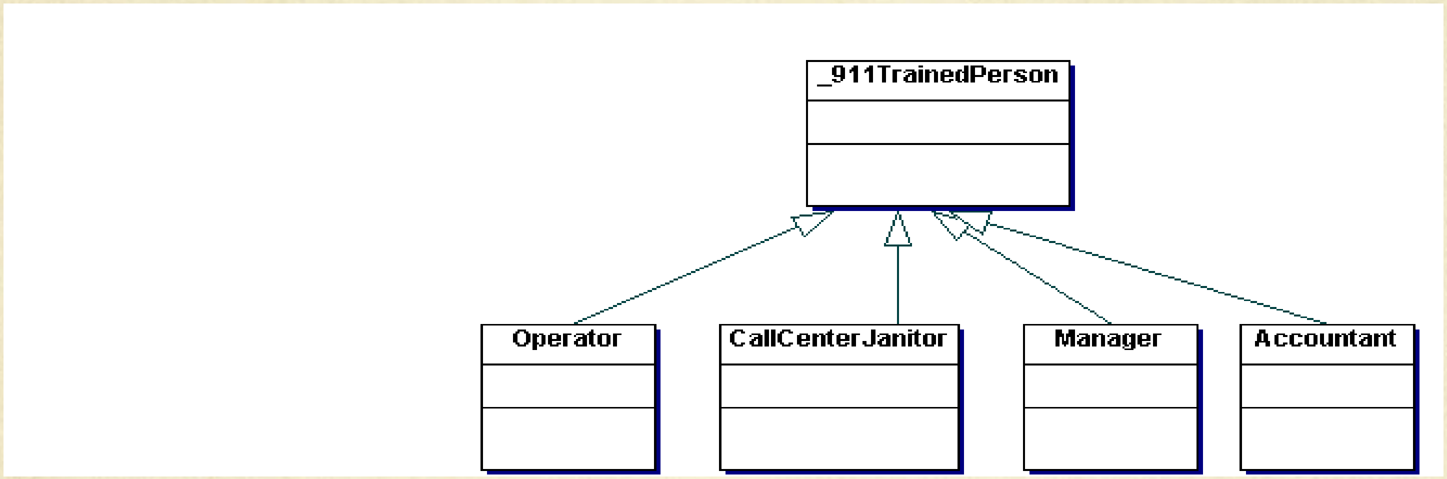
Polymorphism

- Examples of the concept:
  - Huckleberry Finn and the Grangerfords and Sheperdsons
  - Bluto (Belushi) and Otter in Animal House: Delta house versus Omega House, after being beat up, Bluto says: "What … happened to the Delta I used to know? Where's the spirit? Where's the guts, huh? Ooh, we're afraid to go with you Bluto, we might get in trouble… Not me! Not me! I'm not gonna take this. Wormer, he's a dead man! Marmalard, dead! Niedermeyer, dead, Greg, dead…"
  - A fire call to whoever is at the 911 departmental desk: Fire Chief, Lieutenant, Sergeant, Office, Maid, etc.

Polymorphism

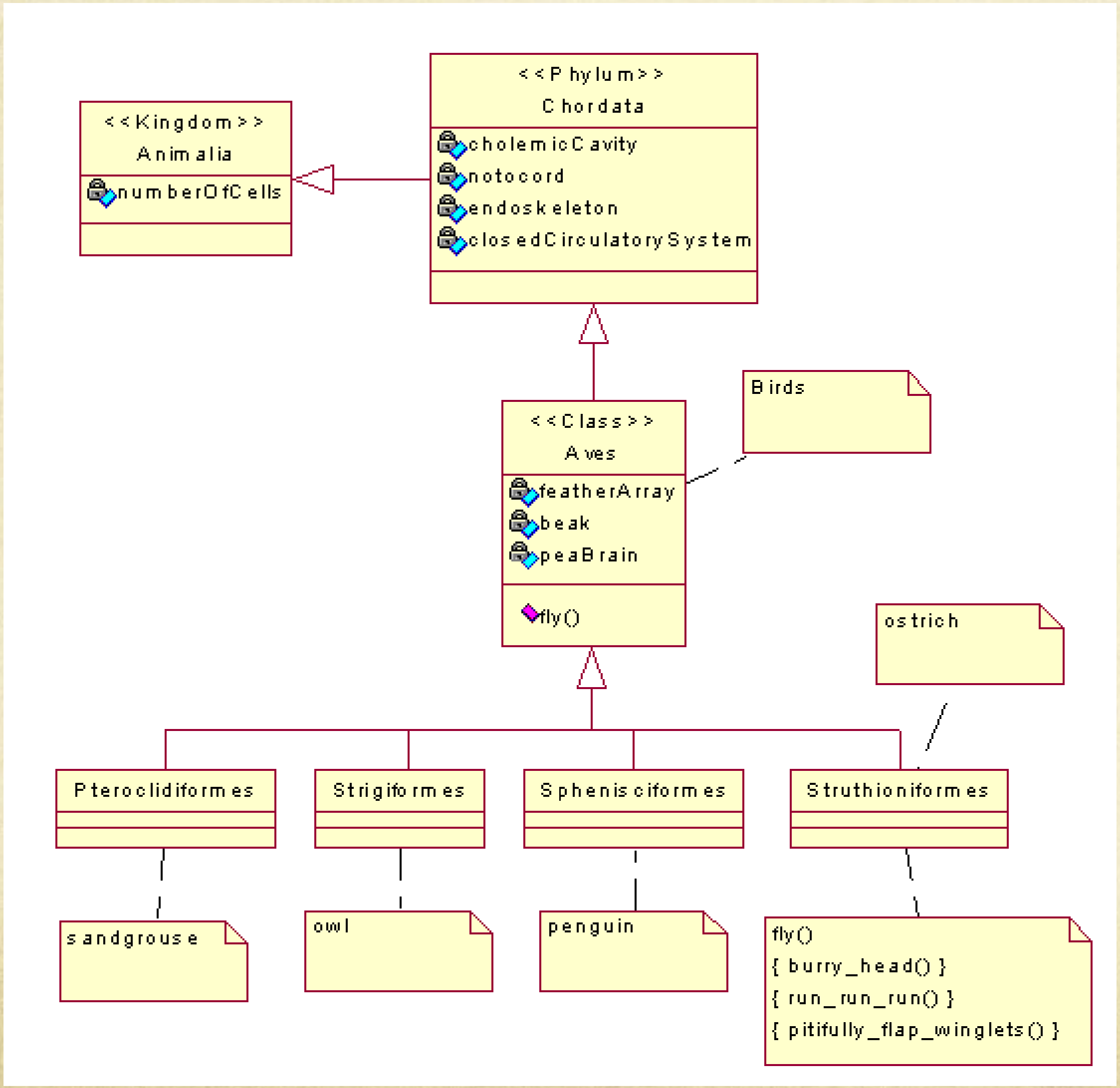
- Polymorphism simply means that you can *command* an instance of a *subtype (some type of thing)* by issuing a command on the base class interface without having to know or care about the *specific subclass type*
- Polymorphism is therefore not politically-correct (it completely removes concern for individuality)
    - Private! Pick up that M60 and head up that hill!

So what does this mean?



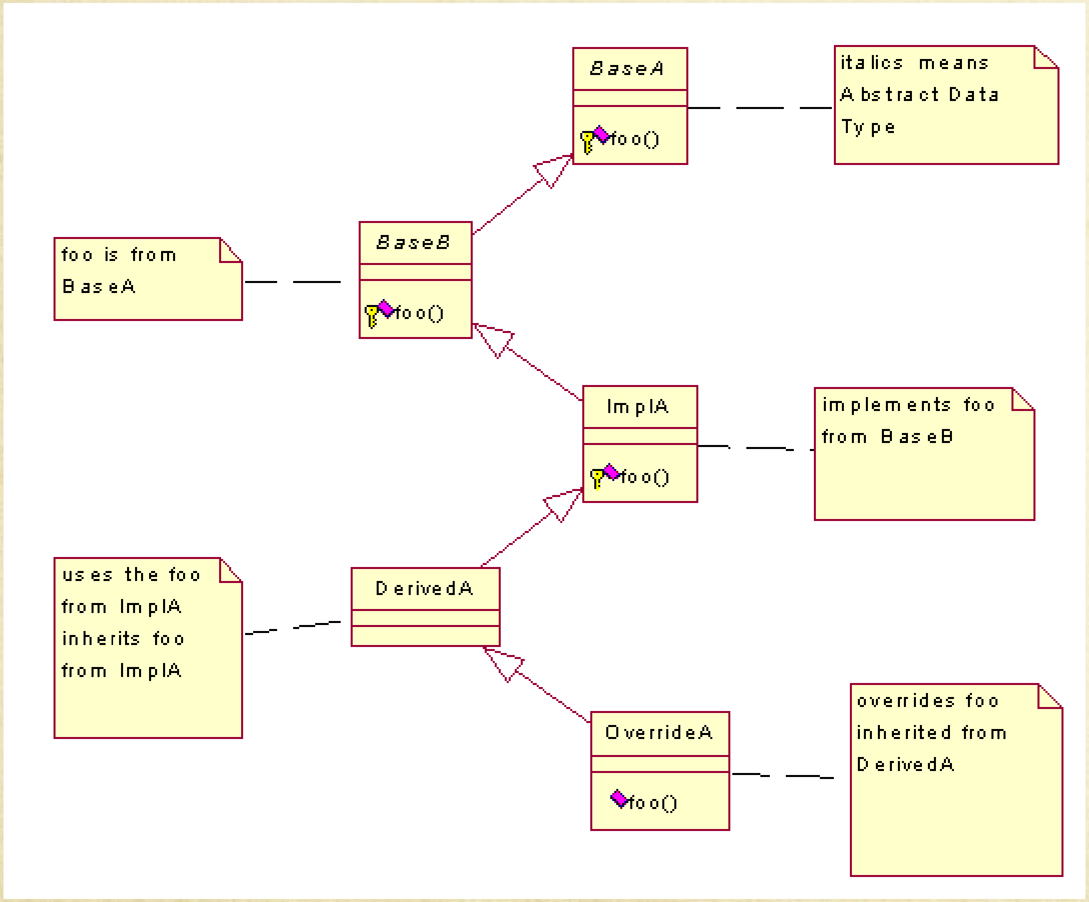
- *Every* employee of the 911 Call Center knows how to handle a 911 call, *regardless of "who they are"* in the organization

Polymorphism Revisited



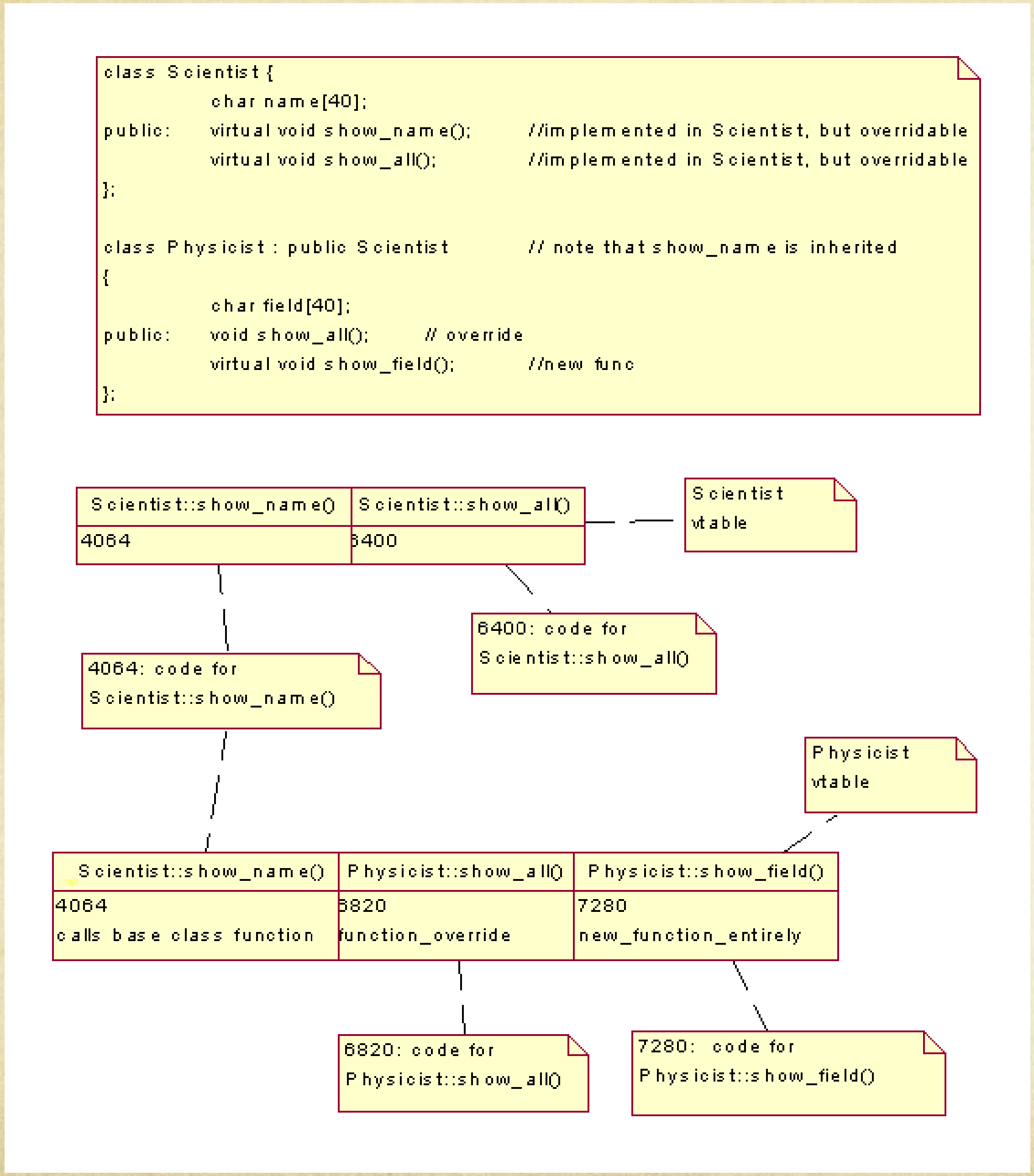
- $(\forall x)(Bx \rightarrow Fx)$
- All Birds Fly
- An Osterich is a Bird
- An Osterich flies?

Essential Polymorphism



BaseA
foo()
italics means Abstract Data Type
BaseB
foo()
foo is from BaseA
ImplA
foo()
implements foo from BaseB
DerivedA
uses the foo from ImplA inherits foo from ImplA
OverrideA
foo()
overrides foo inherited from DerivedA

- A client has a reference to BaseA, instance could be BaseA, ImplA, DerivedA, or OverrideA
- A client has a reference to ImplA, instance could be ImplA, DerivedA, or OverrideA
- A client has a reference to DerivedA, instance could be DerivedA, or OverrideA
- A client has a reference to OverrideA, instance could only be OverrideA
- ***Example***: notvirt in Java and C++



```
class Scientist {
            char name[40];
public:     virtual void show_name();       //implemented in Scientist, but overridable
            virtual void show_all();         //implemented in Scientist, but overridable
};

class Physicist : public Scientist          // note that show_name is inherited
{
            char field[40];
public:     void show_all();        // override
            virtual void show_field();       //new func
};
```

Scientist::show_name()
Scientist::show_all()
4064
6400
Scientist vtable
6400: code for Scientist::show_all()
4064: code for Scientist::show_name()
Physicist vtable
Scientist::show_name()
Physicist::show_all()
Physicist::show_field()
4064 calls base class function
6820 function_override
7280 new_function_entirely
6820: code for Physicist::show_all()
7280: code for Physicist::show_field()

Fast Poly, C++ style

- Physicist p = new Physicist("Feynman");
- Scientist * s = &p;
- s->show_all();
- s->show_name();
- ((Physcist)s) ->show_field();
- ***Example***: scientist.cpp

Typing

- Strongly-typed languages ensure each variable has an defined type, and can only reference objects at runtime that belong to that type (or its derivatives). The compiler guarantees that calls will not fail at runtime. (C++, Java, C#)
- Weakly-typed languages use variables that have no *inherent* type associated with them, variable names are *generic* object references, and can refer to *any* object whatever (Eg. Smalltalk, lisp)
- *Example:* myBagBaby in Smalltalk (music.ws)

Binding

- Binding refers to both:
  - when a *type* is bound to a variable or
  - when a *method is dispatched* to an object:
- Strict Early *Type* Binding: C++ language
  - strictly *compile-time* binding
  - *Allegiance* to type…
  - int x; // better be an int or you' re casting…

Early Typing / Late Binding

- Compile-time Typing: Java (*run-time* binding)
  - Compiler determines which *method signature* to call at *compile time* based on type of parameters ("I' m going to call some object' s *put* method that takes in a *Collection* and an *integer*: ???.put(Collection, int)")
  - Virtual Machine determines which *target object* to send the message to based on the actual object referred to *at runtime*.
  - *Example*:
    - Java: Music.java

Late Method Binding

- Late Binding: Smalltalk (*run-time* binding)
  - No *allegiance* to type in variables
  - Two classes, Rectangle and Inventory, both define a method called *size*
    - MyRef := Rectangle new.
    - MyRef := Inventory new.
    - MyRef size. "to which object is *size* dispatched?"
    - What happened to the Rectangle?
    - In Smalltalk, a method is dispatched at runtime according to the *method name* (selector) and the parameter order of its *arguments*
    - *Example*: Smalltalk: Symphony class

Dynamic Typing Revisited: Duck Typing

- If it walks like a duck, quacks like a duck, and has webbed feet, it's a duck (cf. Justice Potter Stewart, in Jacobellis v. Ohio (1964), James Whitcomb Riley, the Hoosier Poet)
- DT is a dynamic typing in which an object's methods and properties determine the valid semantics, rather than its type of class or implementation of a specific interface
  - C# (~~obj is MyClass~~) or Java (~~instanceof(MyClass)~~)
- Therefore, DT focuses on *capabilities* rather than type, and these capabilities are discovered at runtime (dynamically)
- Smalltalk legacy (mybagbaby), C# (4.0), Python, Ruby, Lisp, Scala
- Downsides (Little Red Riding Hood Syndrome)

Binding and Polymorphism

- Hybrid (C++)
  - C++ with virtual function tables (each object has a vptr in the class' s vtable)
  - C++ virtual function polymorphism works only with pointers and references, not composed objects
  - Fast, no hashing is done, polymorphic behavior is determined by direct vtable offsets
  - pointers and references, not objects.
  - *Example*: music.cpp

Hybrid Languages

- Some languages offer both early and late binding, at the *direction* of the programmer (C++, C#, Eiffel)
- When used polymorphically, the compiler determines which method to call based on *both* the *operation name and* the *static types* of the *parameters*
- *Example:* Music.cs (new versus virtual/override)

Multiple Dispatch (Multi-Methods)

- Some languages not only do not type variables, but methods do not *belong to* classes *per se*, but are generically defined in the environment
- MD languages decide on an implementation of a method ***only*** *at runtime*, based on the actual *types* of parameters of a runtime call
- Thus, depending on the *types* of parameters *discovered* at runtime, the best method *to apply to an object will be selected* from any number of relevant choices
- Thus, a lisp class *has* attributes (slots) but methods are *applied* to a lisp object *at runtime*
- In such languages, the notion of object *self* is missing
- *Examples:* Lisp: music.cl

Binding and Polymorphism

- Pure Polymorphism (Java Interface, C++ ABC)
  - interface vs. implementation inheritance
  - a pure polymorphic method is one which does not have a base implementation
  - a pure virtual function (or an ABC in C++) provides a "placeholder" method name, but does not provide an implementation because the method is *generically* meaningless
  - a "taste" method on a fruit abstract class...

Covariance and Contravariance

- Covariance implies converting from a specialized type (Dandy Dinmont) to a more general type (Terrier): Every Dandy Dinmont is a terrier. [Broadening]
  - Covariance = Specialized -> General (think *up*)
- Contravariance implies converting from a general type (Shapes) to a more specialized type (Rectangle). [Narrowing]
  - Contravariance = General -> Specialized (think *down*)
- C# Delegates support Covariance and Contravariance
  - Delegate methods can vary (narrow) on inherited return types
  - Delegate methods can vary (widen) on inherited parameterized types

Covariance and Contravariance

- Example from C# 4.0:
  - Let: `delegate object MyCallback(FileStream s);`
  - Covariance makes the following legal (return type):

    ```
    string SomeMethod(FileStream s);

    //both strings and FileStreams are Objects
    ```
  - Contravariance makes the following legal (parameter type):

    ```
    Object SomeMethod(Stream s);

    //Stream is a base class of FileStream
    ```
  - However, the following is illegal:

    ```
    Int32 SomeMethod(FileStream s)
    ```
    - Because Int32 is a value type, not a reference type, and thus cannot participate polymorphically (autoboxing does not apply in delegates)

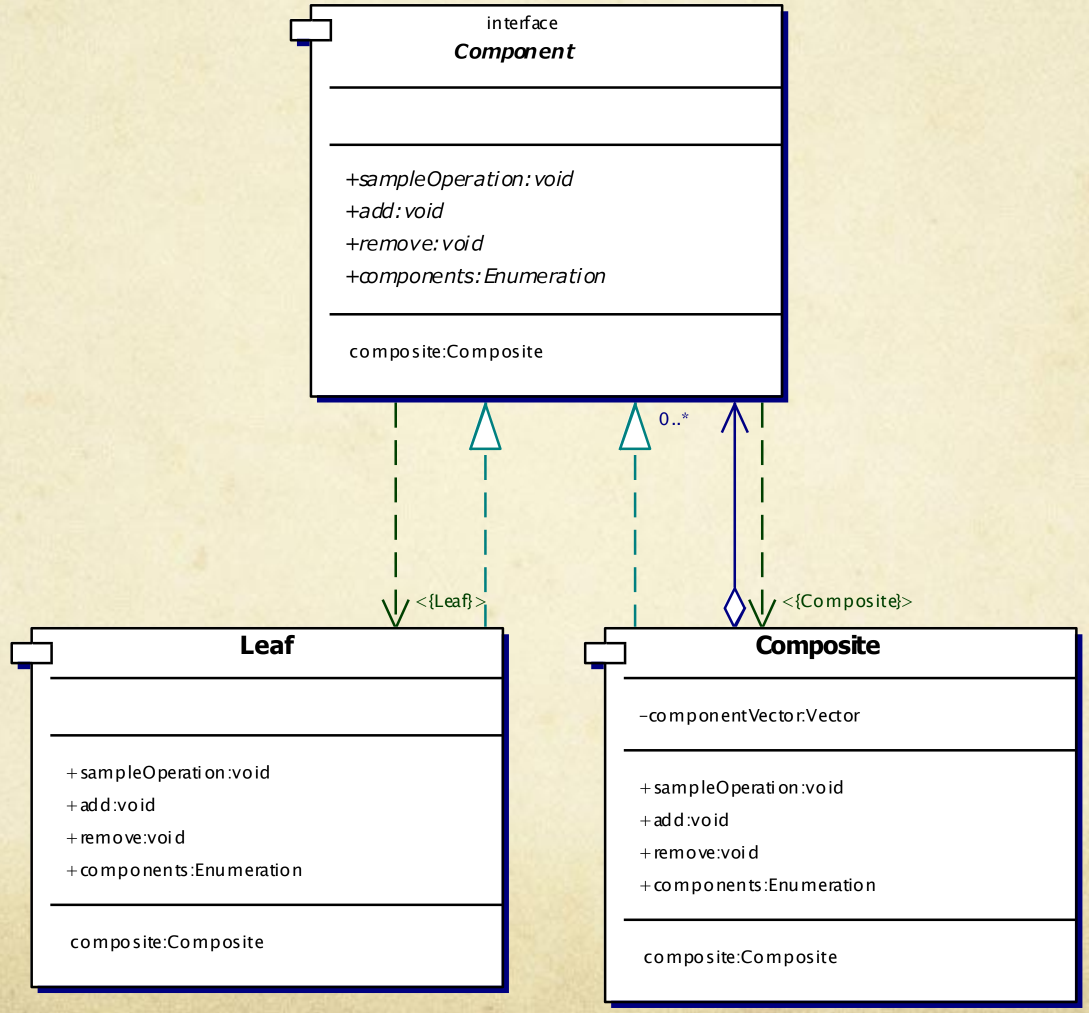
Hallmarks of Good OO Design

- Short methods
- obvious methods (self-evident naming)
- no "God" objects (no one is omniscient)
- no "Manager" objects (nobody is too busy)
- trust your objects to "do the right thing"
  - handle errors through exception interface
- good objects have clear responsibilities
- TELL, don't ask.

Composite

Arrange objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects *uniformly and consistently*.

UML



interface
Component
+sampleOperation:void
+add:void
+remove:void
+components:Enumeration
composite:Composite
0..*
<{Leaf}>
<{Composite}>
Leaf
+sampleOperation:void
+add:void
+remove:void
+components:Enumeration
composite:Composite
Composite
-componentVector:Vector
+sampleOperation:void
+add:void
+remove:void
+components:Enumeration
composite:Composite

Motivation

- An abstract class (Component) represents both the Composite and Leaf nodes
- This allows clients to deal only with the Component interface, and not have to worry about whether they are in reality dealing with a composite structure or a leaf node (unless they want to care).

Benefits

- New classes can be added to an existing component design, i.e., a new leaf type can be added and will automatically work within the existing structure
- New composite types can be added as well
- Clients can traverse a composite structure, and not have to worry about whether they are talking to a leaf node or composite (this is the goal)

Consequences

- Perhaps the biggest decision to make is how to handle the issue of transparency vs. safety. Do we:
  - put an add and remove method (for composites) at the component level;
    - This option values transparency of interface, but problems can arise is someone asks a leaf to add a child
  - put an add and remove method only on the composite where it's actually implemented
    - This option values safety but sacrifices consistency of interface, greatly reducing the core motivation of the pattern in the first place

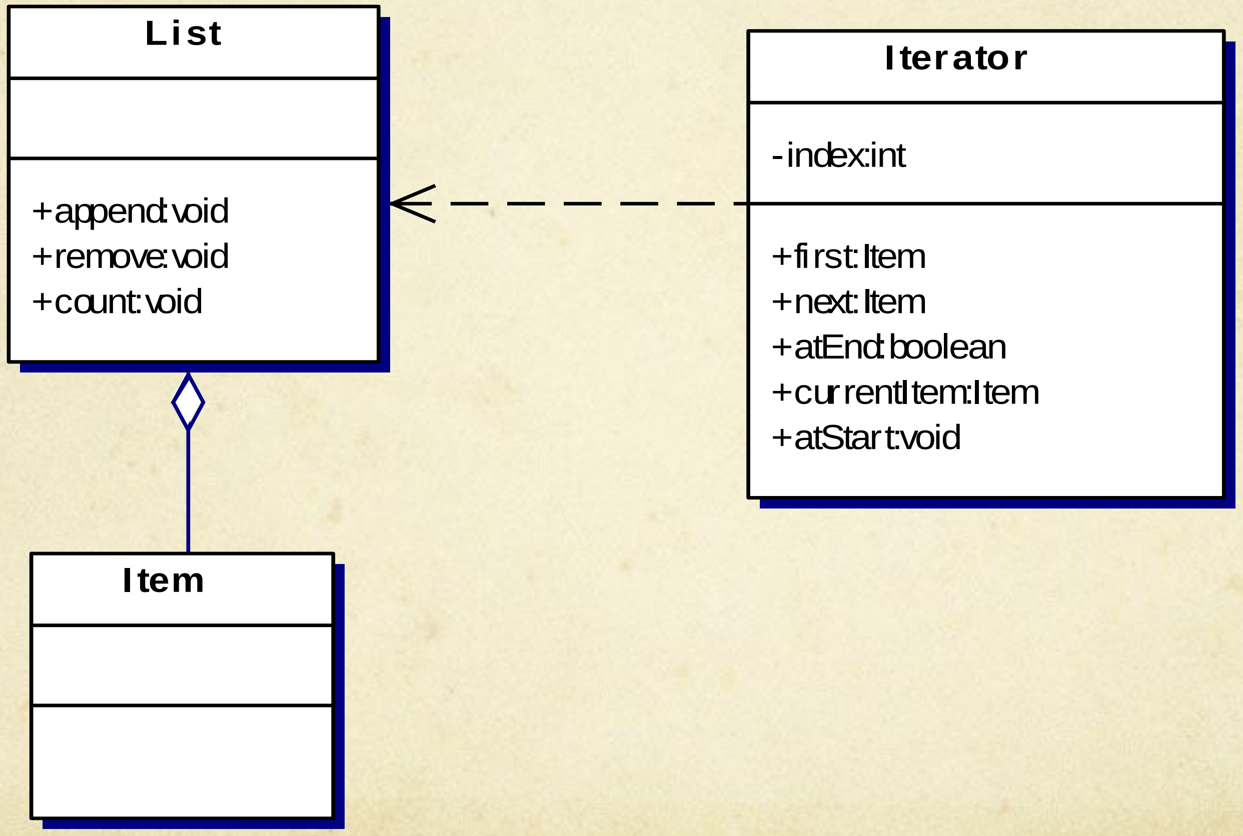
Examples

- Smalltalk
- CLOS
- C#

Iterator (Navigator)

Abstract from a list the methods of access to internal elements, so that the underlying representation of the list is unexposed.

UML


List
+append:void
+remove:void
+count:void
Iterator
-index:int
+first:Item
+next:Item
+atEnd:boolean
+currentItem:Item
+atStart:void
Item

Motivation

- There is a desire to avoid potentially bloating list classes themselves with all the various strategies for walking those lists
- Iterators allow you to abstract out of the lists the strategies for their traversal

Benefits

- By abstracting out the means of traversal, list classes can remain crisp abstractions as pure containers with rudimentary traversal logic
- List classes do not need to be modified just when a new method of traversal appears
- Discrete iterators encapsulate their own cursors, so that simultaneous iterations of the same list by different and multiple iterators is possible

Examples

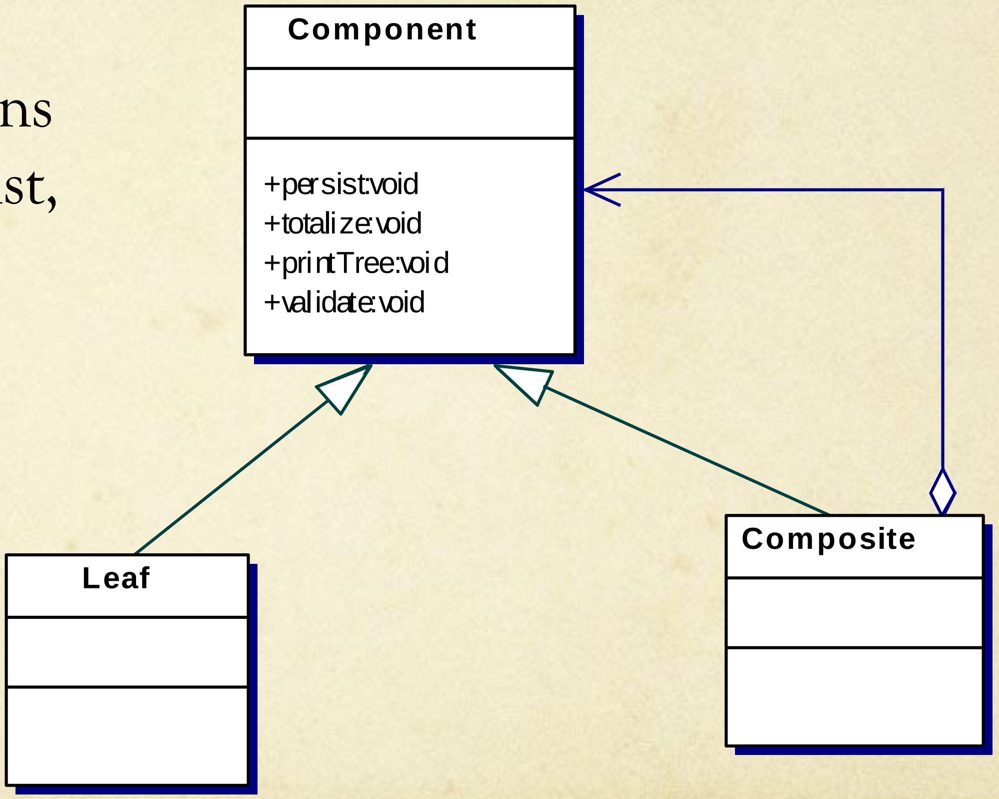
- C#

Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation (Scenario One)

- A class hierarchy whereon a number of discrete and unrelated utility operations must be performed (persist, totalize, copy, instantiate, print, dump, etc.)



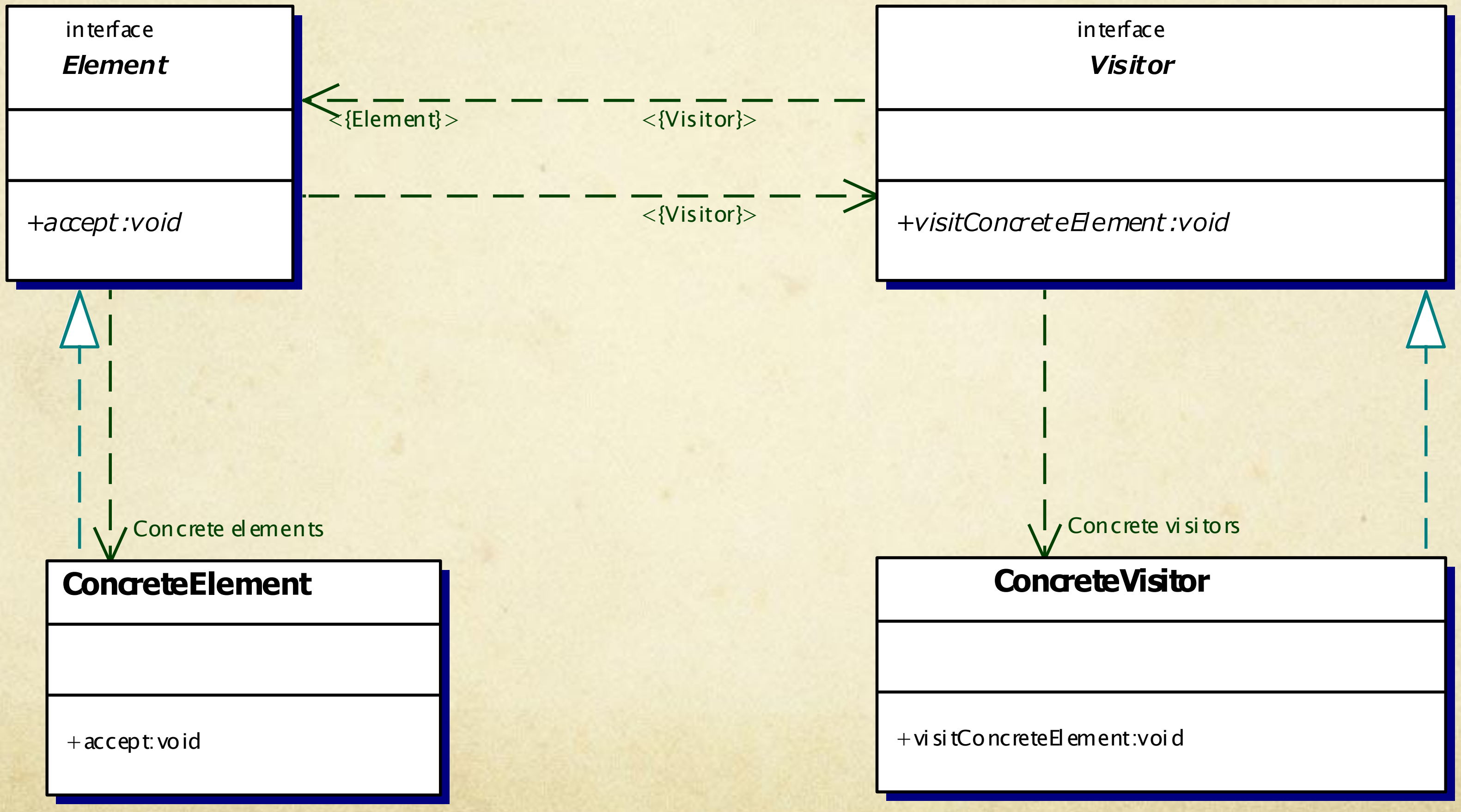
Motivation (Scenario Two)

- A class hierarchy where a number of new operations is expected to be added, but we don’t know these operations at the moment (they are TBD for whatever reason)

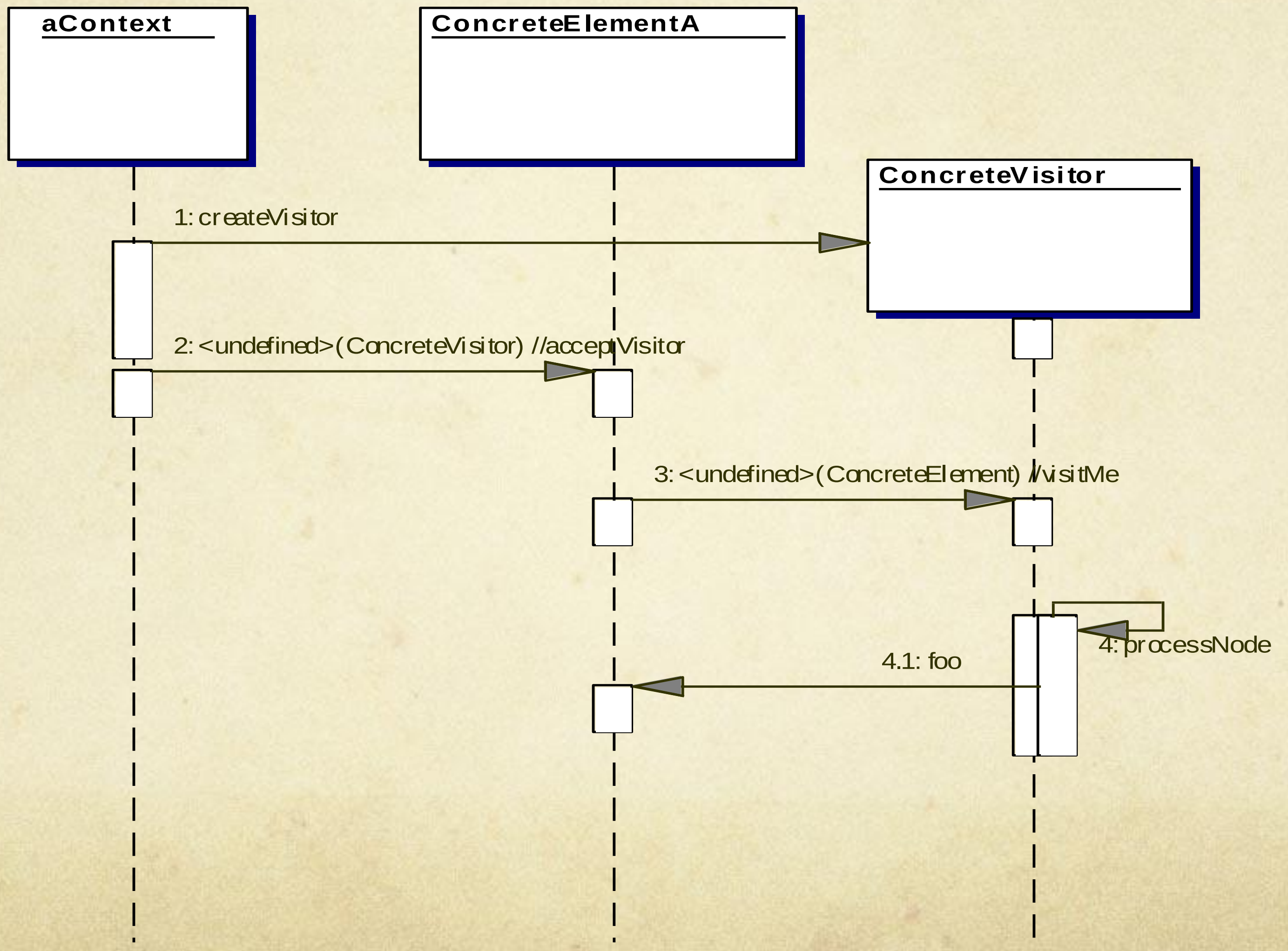
Benefits

- By abstracting (*unencapsulating*) an operation from its class into an external class Visitor hierarchy, we now receive the benefit of dynamically adding new operations as needed
- Related behavior is encapsulated into its own class
- Encapsulation based on *behavior* as opposed to *type*
    - i.e., you wind up with a one or more Persistor visitors rather than a persist() method in many node classes

UML


interface
Element
+accept :void
interface
Visitor
+visitConcreteElement :void
<{Element}>
<{Visitor}>
<{Visitor}>
Concrete elements
Concrete visitors
ConcreteElement
+accept:void
ConcreteVisitor
+visitConcreteElement:void

Sequence


aContext
ConcreteElementA
ConcreteVisitor
1: createVisitor
2: <undefined>(ConcreteVisitor) //acceptVisitor
3: <undefined>(ConcreteElement) //visitMe
4: processNode
4.1: foo

Examples

- C#

Strategy

A Strategy defines a family of encapsulated algorithms, and lets them vary interchangeably independent of the the clients that use them

Motivation

- Anytime you have multiple methods of accomplishing a task, for example, multiple ways to encrypt, you need to be able to switch your method of encryption, and this is traditionally done in some form of "if" or "switch" statement
- This is messy because it hard-codes the available algorithmic solutions into a "manager" clause that makes maintenance and alteration difficult

```
{
  if cipher = "DES3"
     des3encryption();
  else if cipher = "BLOW"
     fishencrypt();
  else if cipher = "HP1770"
     hp1770->encrypt();
}
```

Client code…

The Solution

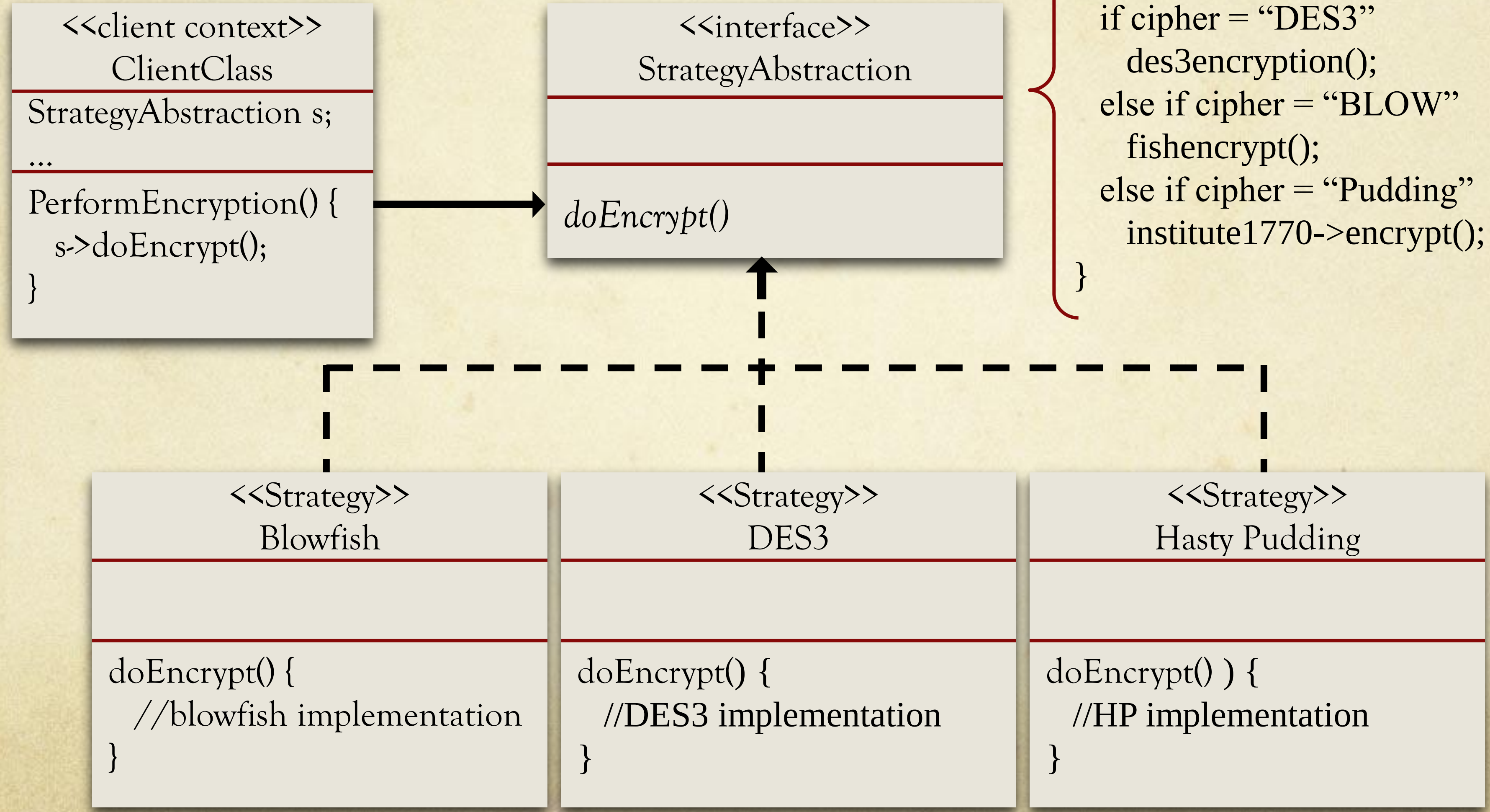
- Define an abstract interface that just defines the interface for the various methods of doing something, effectively replacing the client's "if" logic. This is the abstract interface of a *Strategy*.
- Define an abstract method that is generic on the interface, such as "doEncrypt()". This is the abstract strategy method.
- Define a concrete classes that encapsulate the various methods by which something can be done (the various strategy implementations).
- Implement each independent strategy in its own concrete class, thus making each implementation a *type of Strategy.*

UML

Program to an interface, not to a concrete implementation

A strategy replaces traditional client "if" logic, such as:

```
{
  if cipher = "DES3"
    des3encryption();
  else if cipher = "BLOW"
    fishencrypt();
  else if cipher = "Pudding"
    institute1770->encrypt();
}
```



```
<<client context>>
ClientClass
---
StrategyAbstraction s;
...
---
PerformEncryption() {
  s->doEncrypt();
}
```

```
<<interface>>
StrategyAbstraction
---
---
doEncrypt()
```

```
<<Strategy>>
Blowfish
---
---
doEncrypt() {
  //blowfish implementation
}
```

```
<<Strategy>>
DES3
---
---
doEncrypt() {
  //DES3 implementation
}
```

```
<<Strategy>>
Hasty Pudding
---
---
doEncrypt() ) {
  //HP implementation
}
```

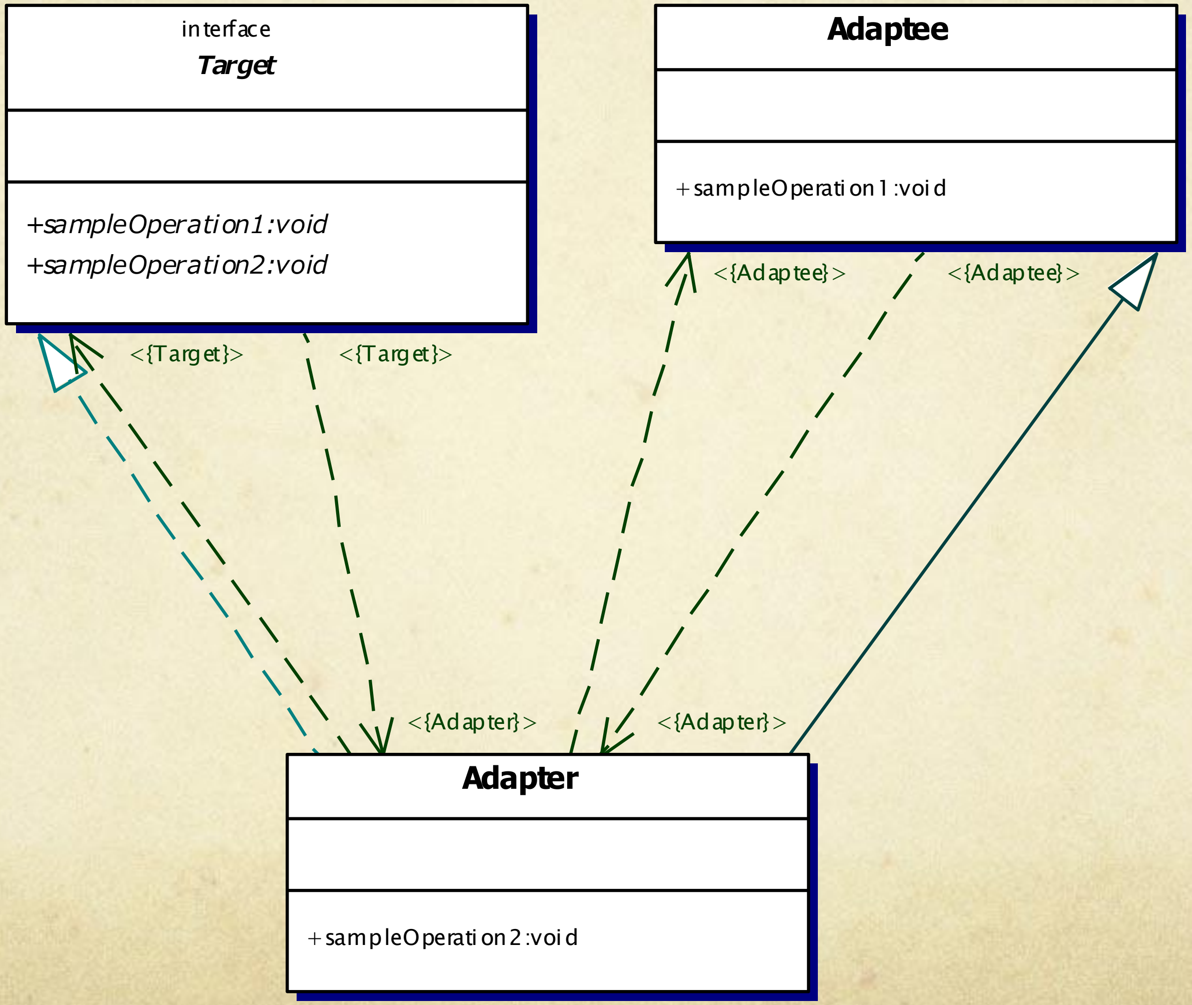
Examples

- Ruby (1 and 2)
- C#

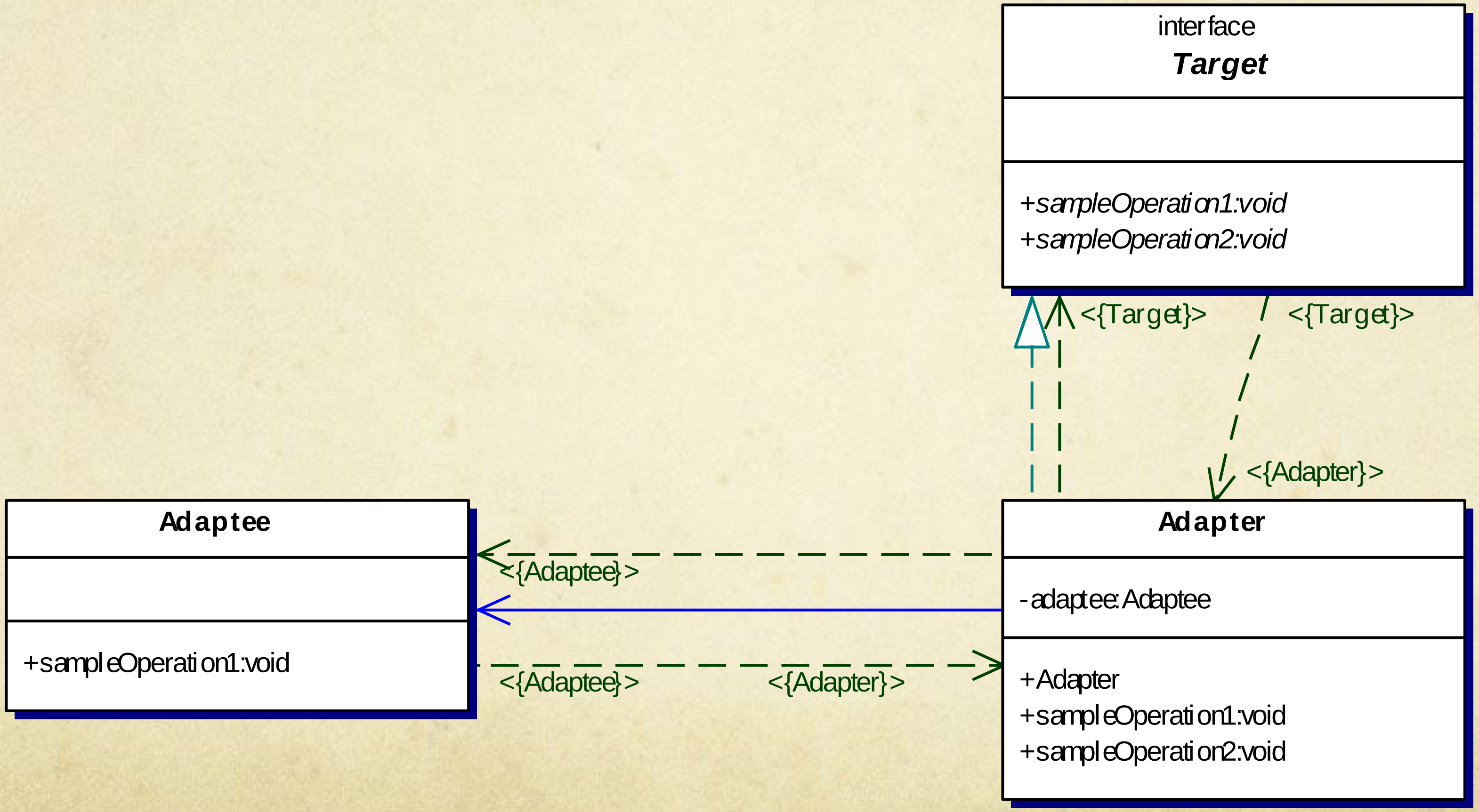
Adapter

The Adapter Pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

UML (Class Adapter)



UML (Object Adapter)



Motivation

- An Adapter is used when existing client calls on one interface need to be redirected to a new interface, without changing the client code
- An Adapter facilitates this by imposing a middleman to accept the old call interface, while delegating out to some new object for the implementation

Benefits

- Using an Adapter allows the substitution of one implementation of an interface for another without affecting existing client code
- Using an Adapter allows one to reuse legacy systems that depend on otherwise outdated or absent libraries or systems

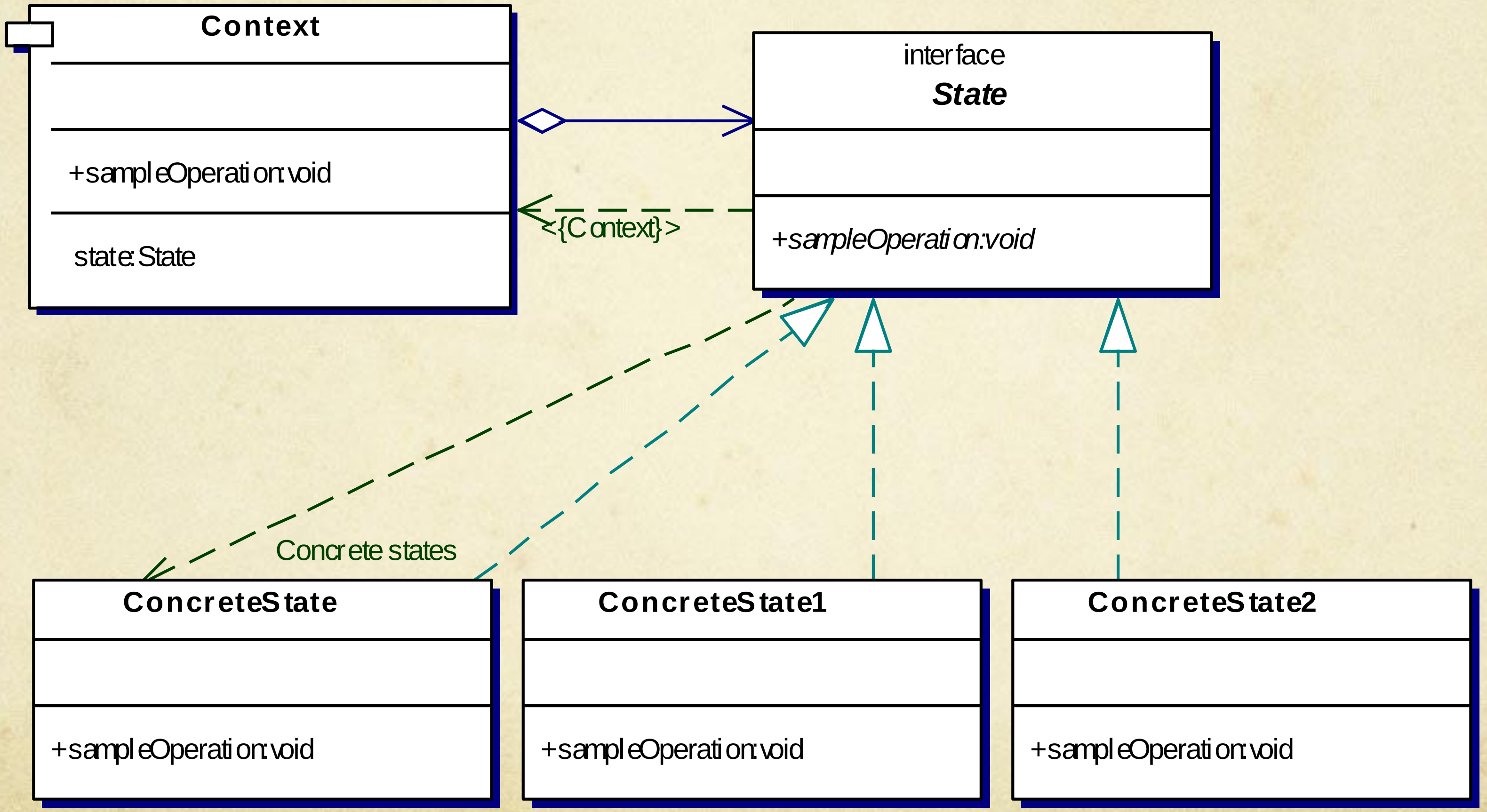
Examples

- C#

State

Model a state machine using objects

UML


Context
+sampleOperation:void
state:State
interface
State
+sampleOperation:void
<{Context}>
Concrete states
ConcreteState
+sampleOperation:void
ConcreteState1
+sampleOperation:void
ConcreteState2
+sampleOperation:void

Motivation

- Traditional state machines are implemented via a global variable representing the state, and accompanying case statements that drive off of the global state
- This is messy because it promotes duplication of code and inflexible design because adding a new state causes sometimes massive code changes, since the state logic is not encapsulated into a class

Benefits

- Because the State pattern encapsulates behavior within individual objects, new states can easily be added by subclassing a new class off of the State class
- State transitions are accomplished by simply rebinding the context variable to point to another state object

Issues

- The pattern leaves it undefined who will initiate state transitions
- Question as to whether states should be instantiated up front or whether they should be instantiated on demand (performance decisions generally drive this)

Examples

- Smalltalk
- C#

Abstract Factory

An *abstract factory* provides an interface for creating families of related objects without specifying their concrete classes.

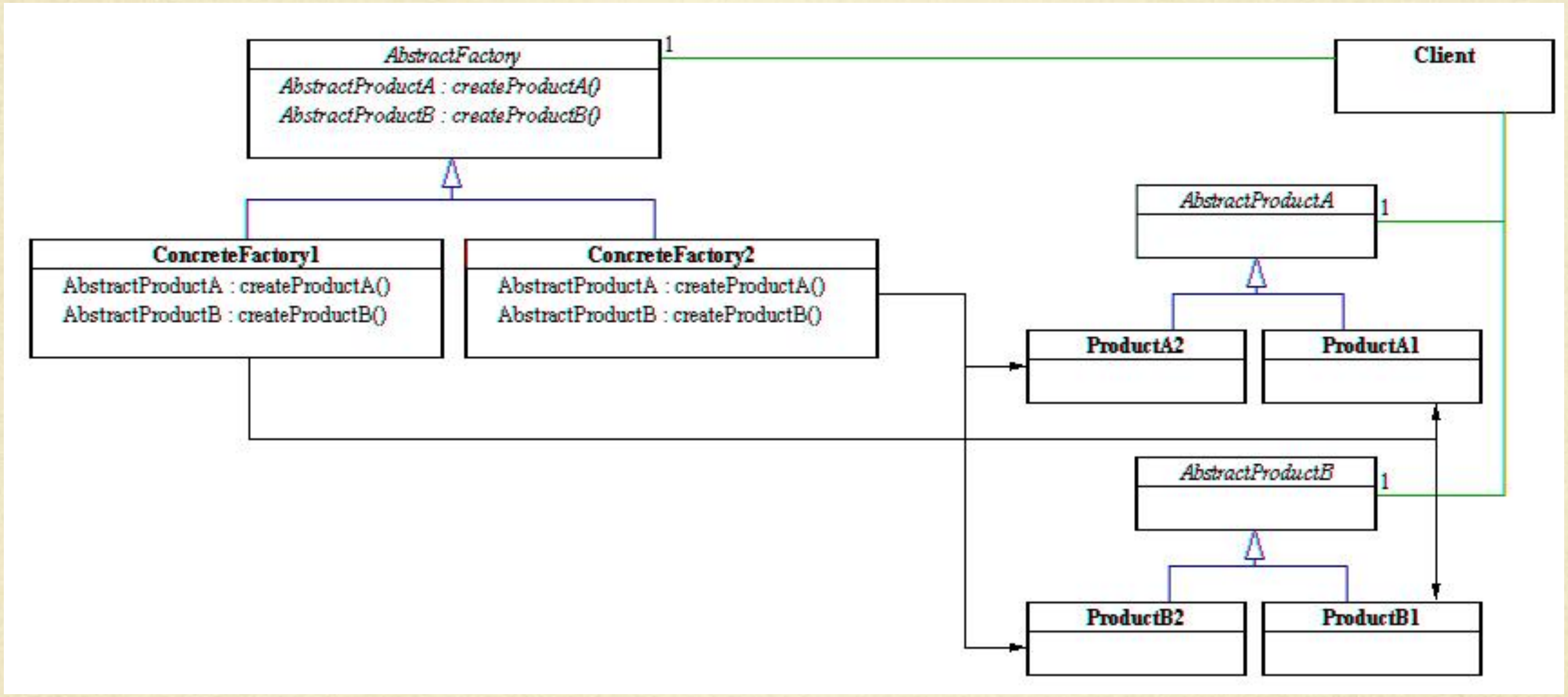
The Problem

- Sometimes, especially in frameworks, you want one of a family of objects created, but:
  - You don’t know what type of object to create, only when to create one (i.e., when asked)
  - You don’t want the client to have to know or care what specific type to create
  - The client cannot know what specific type to create

The Solution

- Define an abstract interface that just defines the interface for creating objects of a given abstract type
- Define a Factory Method that will be used to create actual objects
- Implement classes that implement the factory method so that in effect concrete objects are returned based on the implementations of specific concrete factories
- A concrete factory will specify its products by overriding the factory method for each.
- While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.

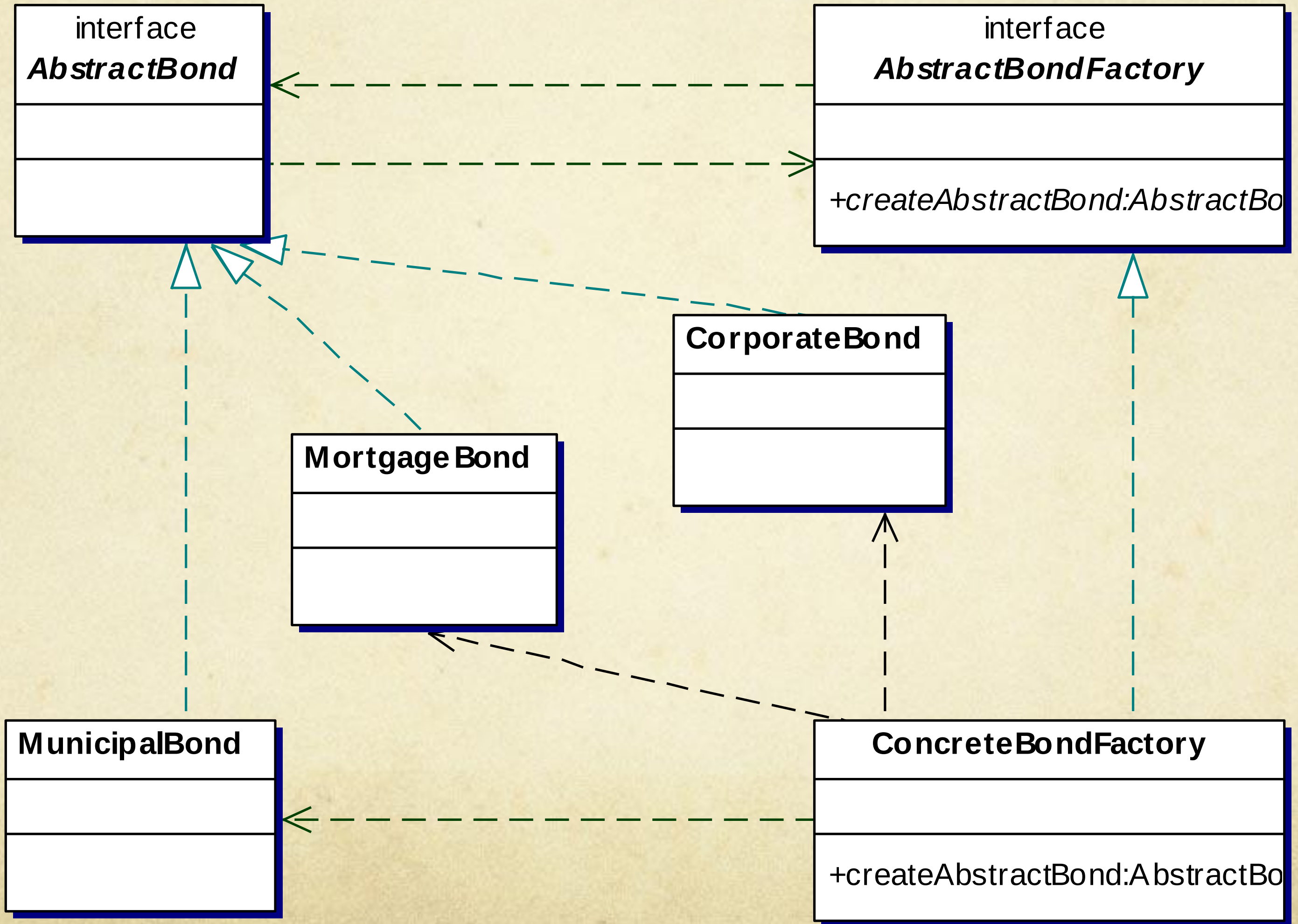
UML


AbstractFactory
AbstractProductA : createProductA()
AbstractProductB : createProductB()
1
Client
ConcreteFactory1
AbstractProductA : createProductA()
AbstractProductB : createProductB()
ConcreteFactory2
AbstractProductA : createProductA()
AbstractProductB : createProductB()
AbstractProductA
1
ProductA2
ProductA1
AbstractProductB
1
ProductB2
ProductB1

Parameterization

- An alternative strategy is based not on subclassing the factory but rather in parameterizing the factory method
- This allows the factory method to key off of some information passed into the method, in deciding which particular type of object to instantiate

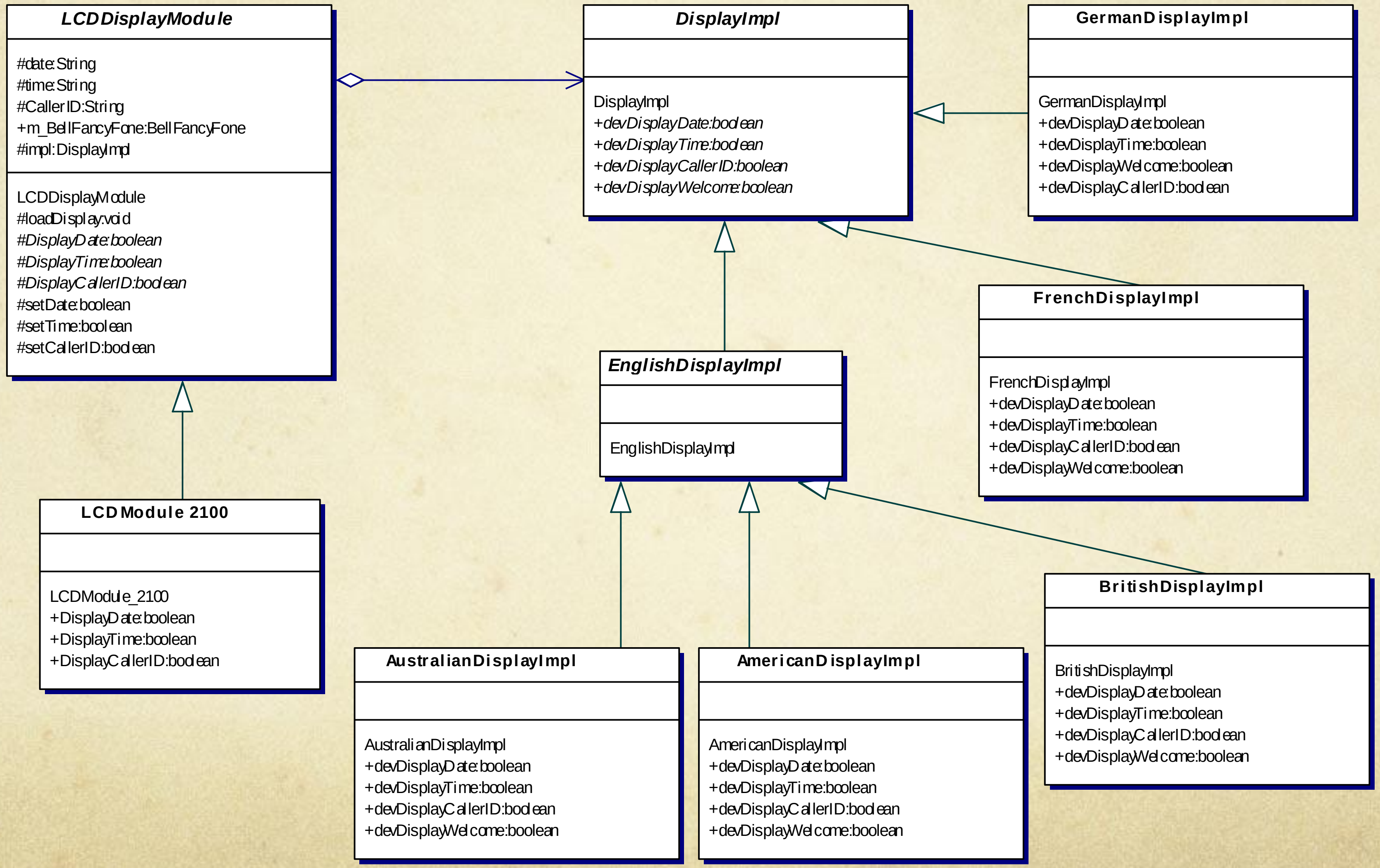
UML


interface
AbstractBond
interface
AbstractBondFactory
+createAbstractBond:AbstractBo
CorporateBond
MortgageBond
MunicipalBond
ConcreteBondFactory
+createAbstractBond:AbstractBo

Bridge

Decouple an abstraction from its implementation so that the two can vary independently

UML


LCDDisplayModule
#date:String
#time:String
#CallerID:String
+m_BellFancyFone:BellFancyFone
#impl:DisplayImpl
LCDDisplayModule
#loadDisplay:void
#DisplayDate:boolean
#DisplayTime:boolean
#DisplayCallerID:boolean
#setDate:boolean
#setTime:boolean
#setCallerID:boolean
DisplayImpl
DisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayCallerID:boolean
+devDisplayWelcome:boolean
GermanDisplayImpl
GermanDisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayWelcome:boolean
+devDisplayCallerID:boolean
FrenchDisplayImpl
FrenchDisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayCallerID:boolean
+devDisplayWelcome:boolean
EnglishDisplayImpl
EnglishDisplayImpl
LCDModule 2100
LCDModule_2100
+DisplayDate:boolean
+DisplayTime:boolean
+DisplayCallerID:boolean
AustralianDisplayImpl
AustralianDisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayCallerID:boolean
+devDisplayWelcome:boolean
AmericanDisplayImpl
AmericanDisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayCallerID:boolean
+devDisplayWelcome:boolean
BritishDisplayImpl
BritishDisplayImpl
+devDisplayDate:boolean
+devDisplayTime:boolean
+devDisplayCallerID:boolean
+devDisplayWelcome:boolean

Motivation

- Inheritance statically binds a particular abstraction (interface) with a particular implementation
- A bridge is used when an abstraction can have one of several possible implementations, but you want to be able to choose dynamically *which* implementation is used

Benefits

- Using a bridge prevents a proliferation of inheritance-based classes (classic example of multiple window platform support)
- Using a bridge allows you to avoid a permanent binding between an abstraction and its implementation
- A bridge allows you to share a single implementation amoung multiple objects, so that the client is unaware of such sharing

Examples

- Java