# Design Patterns

## (**with** Perl **feat.** Moose)

Bryan Smith – March 16, 2011

Ann Arbor Perl Mongers

## Outline

- OO refresher
- Why OO?
- OO design
- Five patterns

## OO refresher: What is OO?

- Paradigm of software design
  - Java, Ruby, Smalltalk
- Other
  - Procedural (C)
  - Functional (Haskell)
  - Logic (Prolog)
  - Domain Specific Languages (SQL, regex)
- Many languages multi-paradigm
  - Perl, C++

## OO refresher: nutshell

- Classes made up of data (attributes) and procedures
- Classes instantiated → objects
- Objects interact to solve programming problem

## OO refresher: benefits

- Encourages encapsulation and modularity
- Polymorphism and inheritance are excellent tools for modeling complex behavior

## Why OO?: vs. procedural

- I prefer OO when...
  - complex runtime behavior
  - large system, need to scale
- I prefer procedural programming when...
  - focus on algorithm
  - alternative to “Utility” classes (OO w/o instantiation)
  - scripts (Unix way)
- Procedural/OO hybrid...
  - algorithm with complex data structures

# OO refresher: `bless` style

```
package Circle;
use Math::Trig;
sub new {
  my $class = shift;
  my $self = { _radius => shift };
  bless $self, $class;
  return $self;
}
sub radius {
  my $self = shift;
  my $radius = shift;
  $self->{ _radius } = $radius if defined( $radius );
  return $self->{ _radius };
}
sub circumference {
  my $self = shift;
  return 2 * pi * $self->radius;
}
sub area {
  my $self = shift;
  return pi * $self->radius * $self->radius;
}
```

## OO refresher: `bless` style

```
package main;

# Make circle
my $circle = Circle->new( 3 );

# Prints: 3 18.8495559215388 28.2743338823081
print $circle->radius . ' ' . $circle->circumference . ' ' . $circle-
>area . "\n";

# Change radius
$circle->radius( 4 );

# Prints: 4 25.1327412287183 50.2654824574367
print $circle->radius . ' ' . $circle->circumference . ' ' . $circle-
>area . "\n";
```

## OO refresher: Moose

```
package Circle;
use Math::Trig;
use Moose;
has 'radius' => (
  is  => 'rw',
  isa => 'Num',
);
sub circumference {
  my $self = shift;
  return 2 * pi * $self->radius;
}
sub area {
  my $self = shift;
  return pi * $self->radius * $self->radius;
}
```

# OO refresher: Moose

```
package main;

# Make circle
my $circle = Circle->new( radius => 3 );

# Prints: 3 18.8495559215388 28.2743338823081
print $circle->radius . ' ' . $circle->circumference . ' ' . $circle-
>area . "\n";

# Change radius
$circle->radius( 4 );

# Prints: 4 25.1327412287183 50.2654824574367
print $circle->radius . ' ' . $circle->circumference . ' ' . $circle-
>area . "\n";
```

## OO design: bad design

- Bad design = *Not the way I would do it*?
- Robert Martin*, bad design not subjective:
  - *Rigidity* – hard to change
  - *Fragility* – unexpected parts of system break
  - *Immobility* – too tangled to reuse

( * “The Dependency Inversion Principle”, Robert C. Martin, C++ Report, May 1996 )

## OO design: **good design**

- What is good design?
  - Recognize it when you see it?
  - good design = not bad design?
- Is good OO design difficult?
  - So many principles and patterns
  - Refactoring OO can be challenging compared to procedural
- Design principles to help prevent bad design

## OO design: Principles (1-2 of 9)

**Open close principle**: Classes should be open for extension but closed for modification.

- *Plain English*: add behavior at runtime

**Dependency inversion principle**: High-level modules should not depend on low-level modules. Both should depend on abstractions.

- *Plain English*: program to an interface

# OO design: Principles (3-4 of 9)

**Interface segregation principle**: Client should not depend on unused interfaces.

- *Plain English*: refactor classes *or* create separate interfaces for different client behaviors

**Single responsibility principle**: Each class should do one thing well.

(Often rephrased as *classes should have one reason to change*.)

## OO design: Principles (5-6 of 9)

**Composite reuse principle**: favor polymorphic composition of objects over inheritance.

- *Plain English*: avoid having to recompile code to change behavior that is likely to change

**Acyclic dependencies principle**: must not be mutual dependencies between classes or packages.

- *Plain English*: if two classes or packages must depend on each other, they probably belong together

# OO design: Principles (7 of 9)

**Liskov's substitution principle**: derived types must be completely substitutable for their base types.

- *Plain English*: subclass behavior should be predictable from the interface alone; shouldn't have to know the type
- aka *Design by Contract*
- *telltale*: violated if any subclass could break assertions or unit test for parent class
- E.g., Square cannot be subclass of Rectangle*

* Robert Martin, "The Liskov Substitution Principle" (1996)

## OO design: Principles (8 of 9)

**Dependency injection principle**: separate dependency resolution from behavior.

- *Plain English*: if a class (e.g., `Vacuum`) needs another class (e.g., `VacuumBag`), use another class (e.g., `VacuumFactory`) to *inject* it.
- Easier to unit test
  - Inject no-op components

## OO design: Principles (9 of 9)

**Inversion of control**: instead of central control of flow of execution (*procedural*), relinquish control and instead focus on one task

- *Plain English*: write code that uses hooks, events, etc. so that anyone can use it.
- *Telltale*: when you don’t get to control when certain things are invoked
- aka *Hollywood Principle*: Don’t call us; we’ll call you.
- Key difference between a framework and a library*

* “InversionOfControl“, Martin Fowler

## OO design: Purpose of patterns

- Reusable, proven solutions (tool set)
- Apply principles
- Vocabulary for developers
- Layer of abstraction (chunking)

## Five patterns

1. Singleton
2. Decorator
3. Adapter
4. Proxy
5. Composite pattern

## Five patterns: 1. Singleton

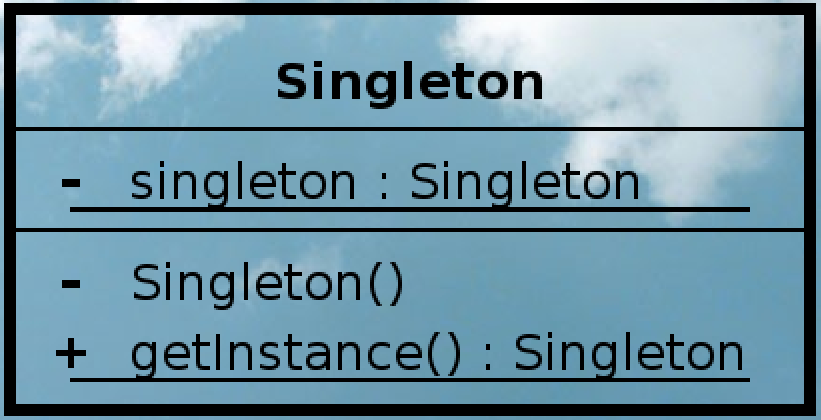
**Problem**: Text editors offer users many preferences (e.g., tabs vs. spaces). We need to store user preferences in a single object that can be globally accessed.

**Solution**: create a `Preferences` class and use singleton pattern so that the same object is returned.

## Five patterns: 1. Singleton

- Provide a global access to a single object
- Accessed like a getter, but returns the same instance for each invocation
- Often combined with **lazy loading**

## Five patterns: 1. Singleton



Source: http://en.wikipedia.org/wiki/File:Singleton_UML_class_diagram.svg

## Five patterns: 1. Singleton

```
#!/usr/bin/perl
package Configuration;
my $singleton;

sub get {
  if ( ! defined( $singleton ) ) {
    print "I'm the only instance!";
    $singleton = _new( @_ );
  }
  return $singleton;
}

sub _new { # < don't directly instantiate!
  my $class = shift;
  my $self = {};
  bless $self, $class;
  return $self;
}

# methods ...

package main;
my $config = Configuration->get();

# later ...
my $sameConfig = Configuration->get();
```

## Five patterns: 1. Singleton

```
#!/usr/bin/perl

package Configuration;
use MooseX::Singleton; # < Instead of “use Moose;”

# methods ...

package main;
my $config = Configuration->new();

# later ...
my $sameConfig = Configuration->new();
```

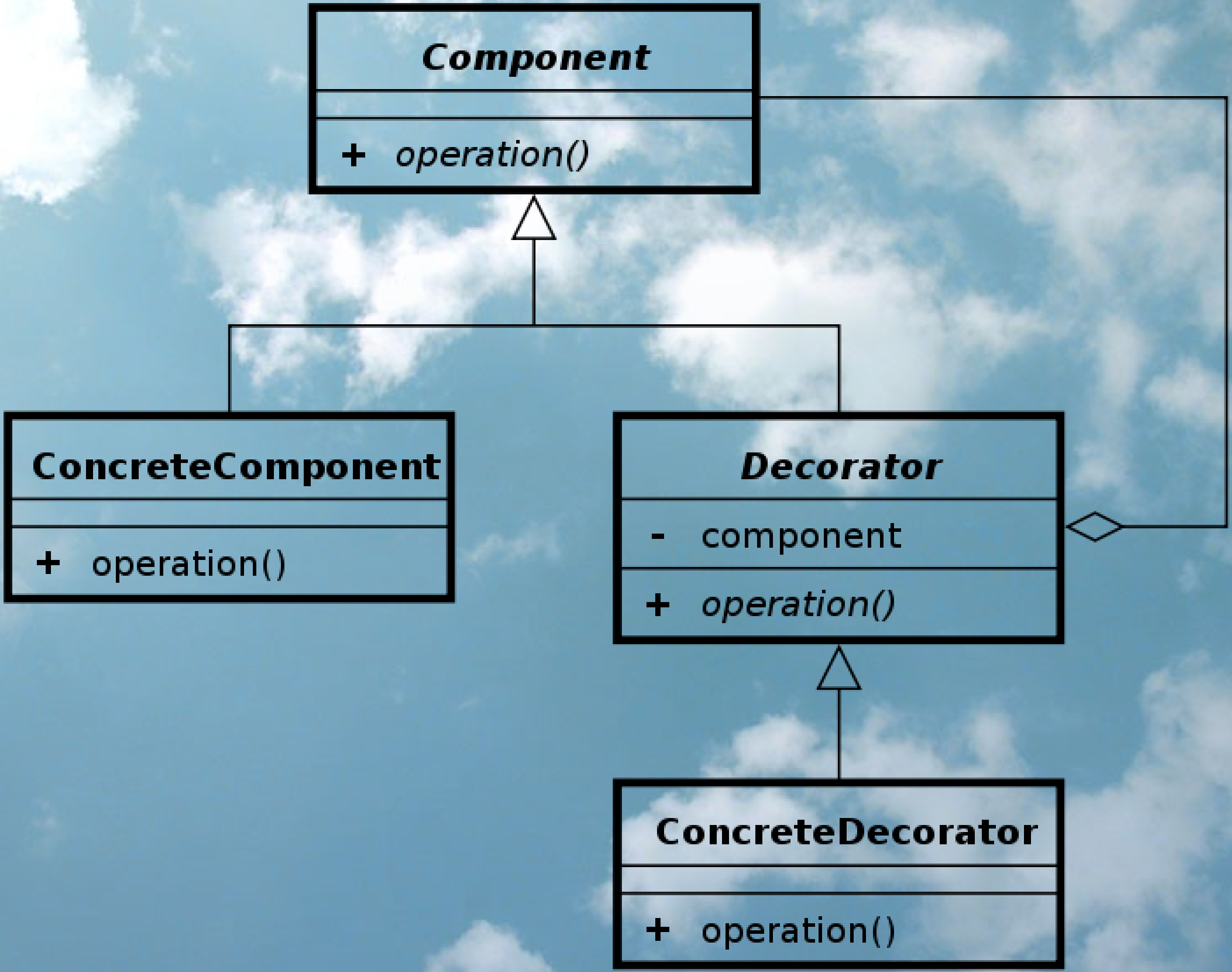
## Five patterns: 2. Decorator

**Problem**: your turn-based game can put a character through a lot. The character can pick up a cursed axe, become poisoned (or doubly poisoned), start to starve, get confused, etc. Your starving elf might get poisoned eating some bad food. How do you program these behaviors (and future behaviors) in a manageable way?

**Solution**: Create a `Character` interface, and implement different characters (e.g., `Human`, `Elf`, `Ogre`, `Vampire`). Then use the decorator to wrap these implementations with additional behavior.

## Five patterns: 2. Decorator

- Wrap an implementation with another class with same interface to add functionality at runtime.
- Forward all behavior on to wrapped class except when decorator should prevent invocation.
  - a `PoisonedPlayer` can still walk, but will suffer damage.
  - but a `PlayerWithBrokenLeg` cannot walk (unless wielding a staff)



Source: http://en.wikipedia.org/wiki/File:Decorator_UML_class_diagram.svg

## Five patterns: 2. Decorator

```
package Player;
use Moose::Role;

requires 'walk';
requires 'loseLife';

# Other behavior like eat, pickup...
```

## Five patterns: 2. Decorator

```
package Human;
use Moose;

with 'Player';

sub walk() {
  my ($self, $direction ) = @_;
  print "The human walks ${direction}.\n";
  # walk behavior...
}

sub loseLife() {
  my ( $self, $damage ) = @_;
  print "Ouch! You take $damage damage.\n";
  # damage behavior...
}

# Other req. behavior...
```

## Five patterns: 2. Decorator

```
package PoisonedPlayer;
use Moose;

with 'Player'; # < Note that the wrapper can be wrapped itself!

has 'player' => (
  does => 'Player', # < Only wrap something with Player role
  is => 'rw',
  required => 1,
);

sub walk() {
  my $self = shift;
  $self->loseLife( 2 );       # < Lose a little life first
  $self->player->walk( @_ ); # < Defer to wrapped class
}

sub loseLife() {
  my $self = shift;
  $self->player->loseLife( @_ ); # < Just forward it along
}

# Other req. behavior...
```

## Five patterns: 2. Decorator

```
package main;

my $human = Human->new();

# "The human walks south."
$human->walk( 'south' );

my $poisonedHuman = PoisonedPlayer->new( 'player' => $human );

# "Ouch! You take 2 damage."
# "The human walks east."
$poisonedHuman->walk( 'east' );

# Doubly poisoned
my $veryPoisonedHuman = PoisonedPlayer->new( 'player' =>
$poisonedHuman );
```

## Five patterns: 3. Adapter

**Problem**: Your application has its own authentication system, but you’d also like to allow your users to log in using a third-party service.

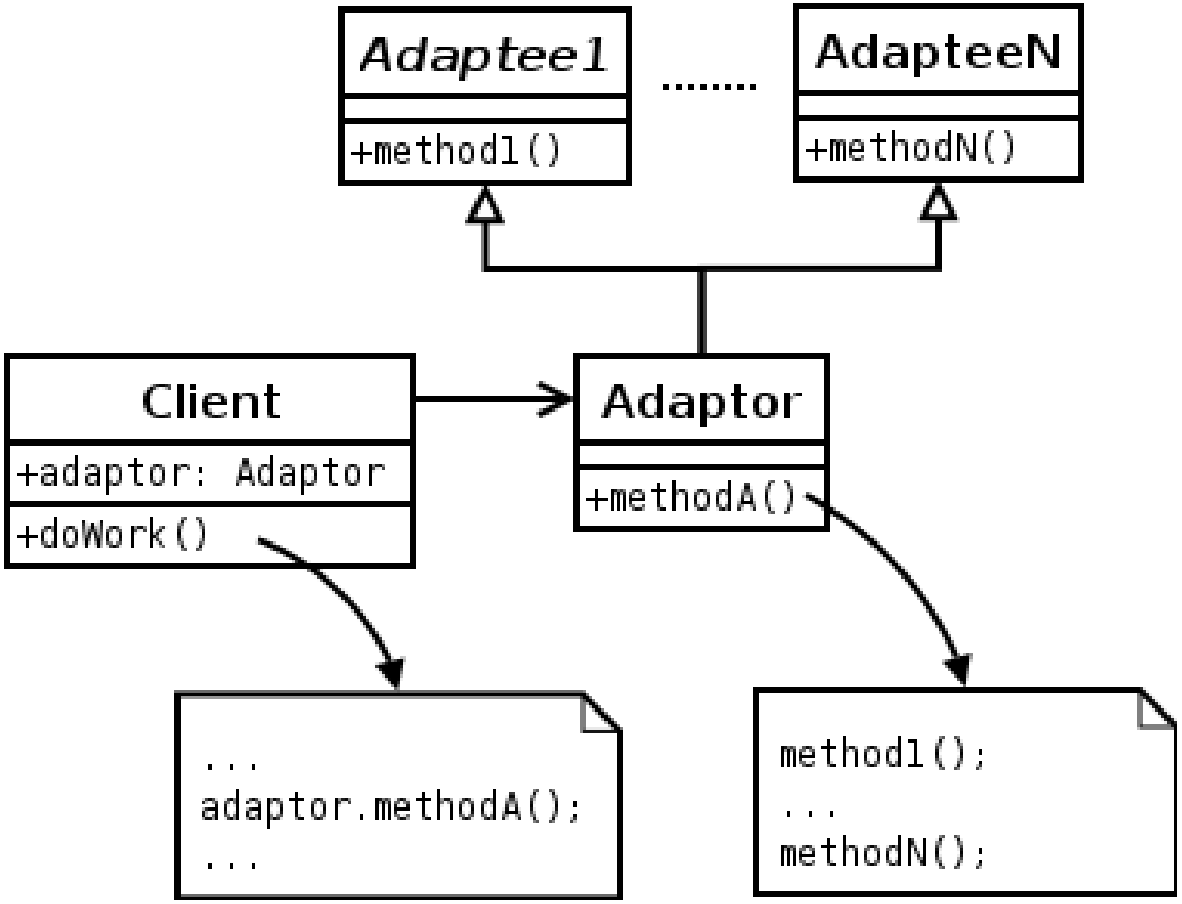
The problem is that Yapoo's service returns an authentication object with a different interface.

**Solution**: Create an adapter for the third-party objects.

## Five patterns: 3. Adapter

- Wrap a class (which has an incompatible interface) to give it a compatible interface.

## Five patterns: 3. Adapter



Source: http://en.wikipedia.org/wiki/File:Decorator_UML_class_diagram.svg

## Five patterns: 3. Adapter

```
package User;
use Moose::Role;

requires ‘name’;
requires ‘email’;

# - - - - - - - - - - - - -

package UserImpl;
use Moose;

# implement required functionality...
```

## Five patterns: 3. Adapter

```
package UserAdapter;
use Moose;
with ‘User’;

has ‘theirUser’ = (
  isa      => ‘YapooUser’, # < You don’t control this interface...
  is       => ‘rw’,
  required => 1,
);

sub name() {
  my $self = shift;
  return $self->theirUser->uniquename(); # hmmm...
}

sub email() {
  my $self = shift;
  return $self->theirUser->email();      # better way?
}
```

## Five patterns: 3. Adapter

```
package UserAdapter;
use Moose;
with ‘User’;

has ‘theirUser’ = (

  isa      => ‘YapooUser’,
  is       => ‘rw’,
  required => 1,

  # Use delegation
  handles  => {
    name  => 'uniquename', # Much better.
    email => 'email',
  },

);
```

## Five patterns: 4. Proxy

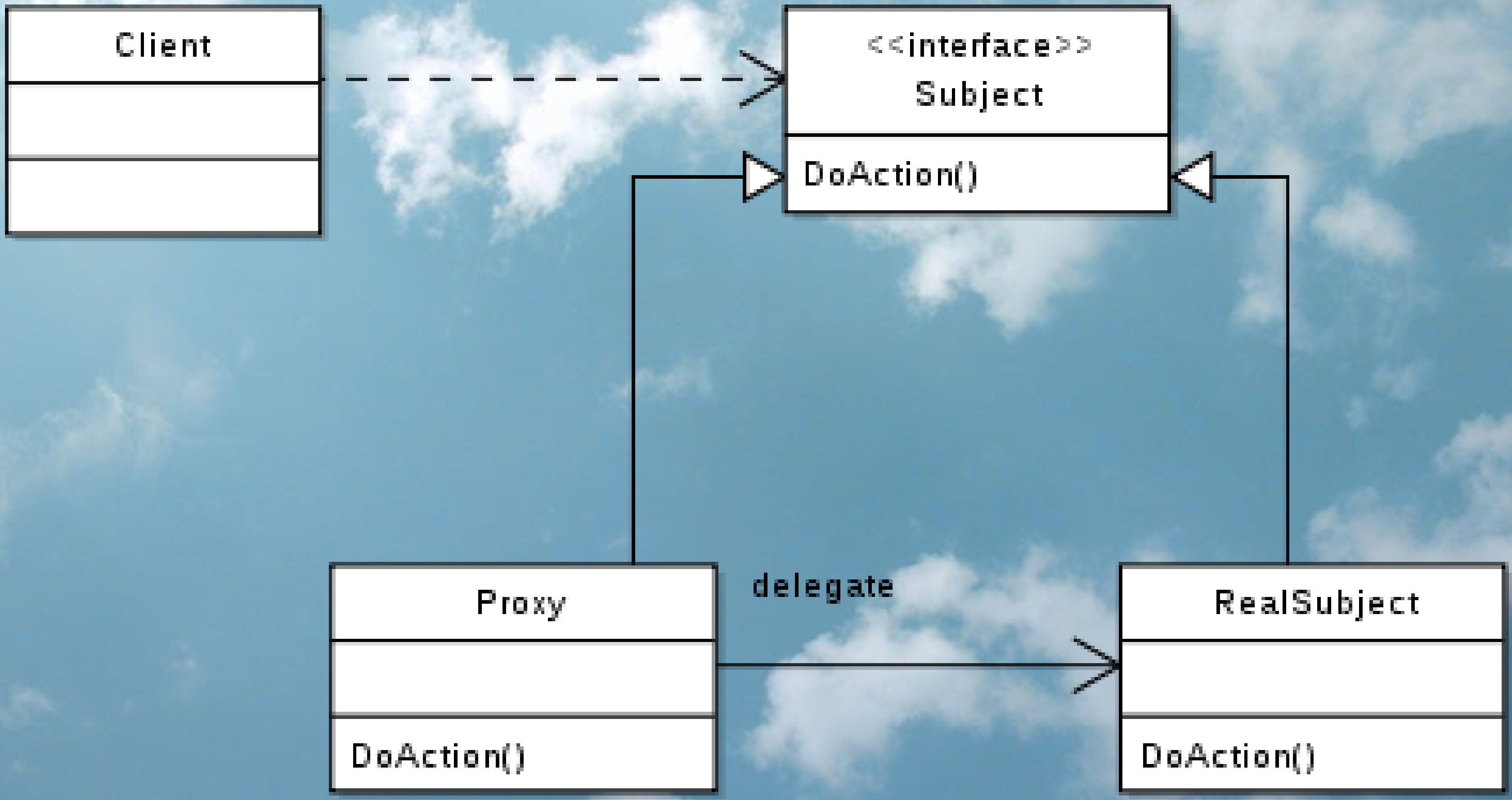
**Problem**: your spreadsheet software needs to read and edit documents stored on a local hard drive or stored remotely.

**Solution**: Create a document proxy with the same interface to communicate over the internet.

## Five patterns: 4. Proxy

- Controls access to an object.
- Since it has the same interface as the object, the user need not know that there is a proxy
- Uses include internet access, security, limiting resource consumption, and more.

## Five patterns: 4. Proxy



Source: http://en.wikipedia.org/wiki/File:Proxy_pattern_diagram.svg

## Five patterns: 4. Proxy

```
package Spreadsheet;
use Moose::Role;

requires ‘open’;
requires ‘close’;
requires ‘getCell’;
requires ‘putCell’;

# - - - - - - - - - - -

package SpreadsheetImpl;
use Moose;

with ‘Spreadsheet’;
# ... implement required functionality.

# Note this is all we need to locally
# manipulate our spreadsheets.
```

## Five patterns: 4. Proxy

```
package RemoteProxySpreadsheet; # The proxy
use Moose;

with Spreadsheet; # < User won't even know it's a proxy!
my $connection;

function open {
  my ( $self, $filename ) = @_;
  my $host = $self->parseHost( $filename );
  $connection = SpreadsheetConnection->connect( $host );
  return $client->open( $filename );
}

# - - - - - - - - - - - -
package SpreadsheetConnection;
use Moose;

# This class handles the socket connection
# with a remote server

# - - - - - - - - - - - -
package SpreadsheetServer;
use Moose;

# Runs on remote server. Talks to Connection
# client. Will instantiate a SpreadsheetImpl
# and serialize & return the results.
```

## Five patterns: 5. Composite

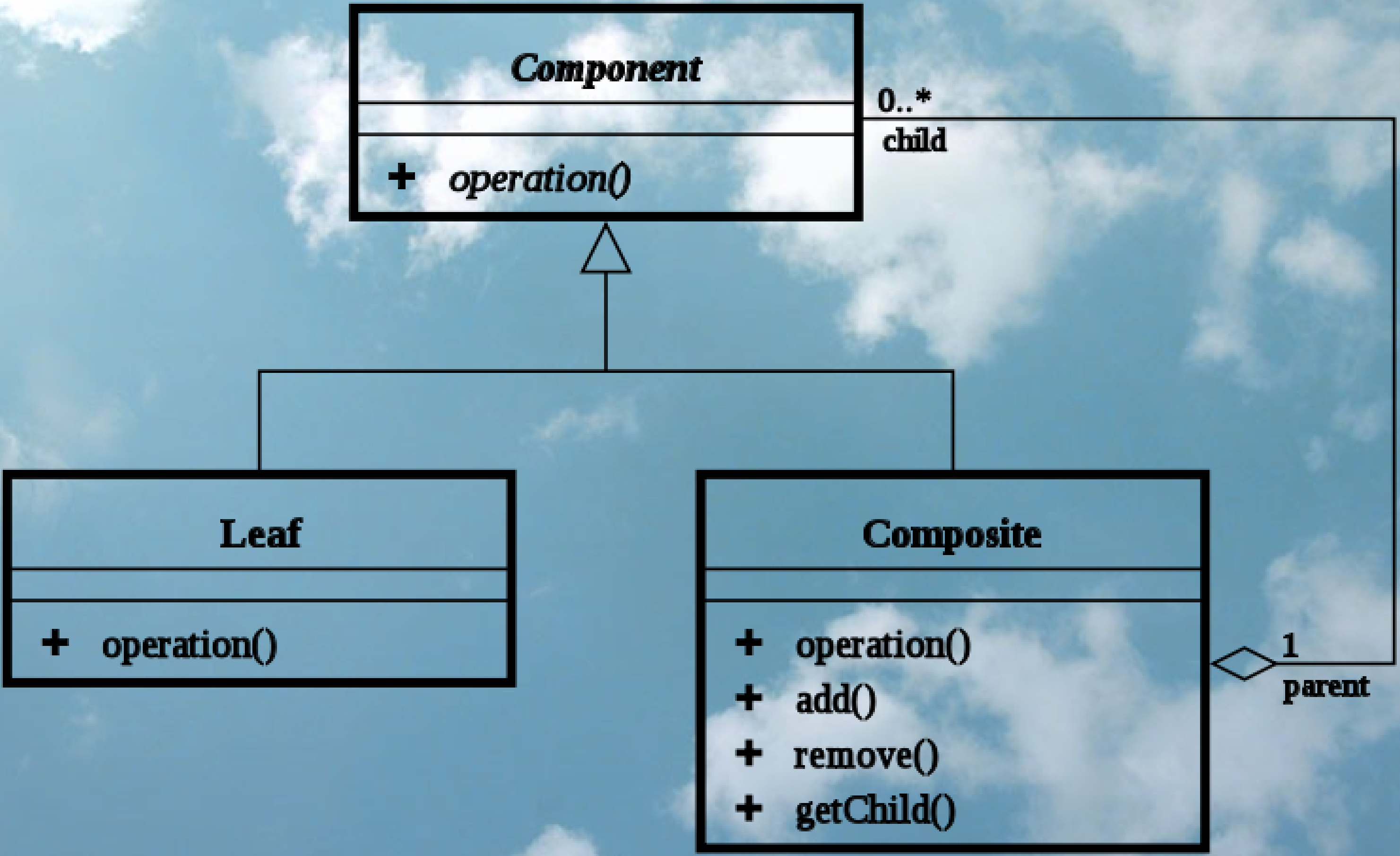
**Problem**: our turn-based game allows bands of warriors to fight together. Furthermore, it allows different groups to temporarily join forces, dealing even more damage per turn.

**Solution**: the composite pattern allows us to arbitrary band together any number of players and groups of players.

## Five patterns: 5. Composite

- Compose objects with the same interface in a tree structure
- Invoking a method for composite will invoke method for all members recursively
- Commonly used for GUIs to recursively repaint widgets when an update has been made

## Five patterns: 5. Composite



Source: http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).svg

## Five patterns: 5. Composite

```
#!/usr/bin/perl

package Player;       # < Same interface from decorator example
use Moose::Role;

requires 'walk';
requires 'loseLife';
requires 'attack';    # < This is the only change

# Other behavior like eat, pickup...
```

## Five patterns: 5. Composite

```
# Some races. Nothing exciting yet...

package Human;
use Moose;

with 'Player';

# required functionality ...

# - - - - - - - - - -
package Elf;
use Moose;

with 'Player';

# required functionality ...

# - - - - - - - - - -
package Ogre;
use Moose;

with 'Player';

# required functionality ...
```

## Five patterns: 5. Composite

```
package Band; # < The composite
use Moose;

with 'Player';                      # 'Band' is a Player, too.

has 'players' => (
  traits  => ['Array'],
  is      => 'ro',
  isa     => 'ArrayRef[Player]', # < Non-players give runtime error.
  default => sub { [] },
  handles => {
    join       => 'push',         # < Another example of delegation
  }
);

sub attack() {
  my $self = shift;
  for my $player ( @{ $self->players } ) { # < Sweet! Massive damage!
    $player->attack( @_ );
  }
}

# required functionality, but each must iterate through 'players' array...
```

## Five patterns: 5. Composite

```
my $human1 = Human->new();
my $human2 = Human->new();
my $poisonedHuman2 = PoisonedPlayer->new( 'player' => $human2 );

my $humans = Band->new();

$humans->join( $human1 );
$humans->join( $poisonedHuman2 );        # < Might be problem later...
$humans->walk( 'north' );

my $cave = environment->get();
my $dragon = $cave->getDragon();         # < Oh no, a dragon!

my $ogre = Ogre->new();
my $elf = Elf->new();

my $unlikelyAlliance = Band->new();
$unlikelyAlliance->join( $ogre );
$unlikelyAlliance->join( $elf );

my $army = Band->new();

$army->join( $humans );
$army->join( $unlikelyAlliance );

$army->attack( $dragon );                # < army attacks the dragon together
```

## Five patterns: notice something?

- **4** out of **5** of these patterns require that you program to an interface!

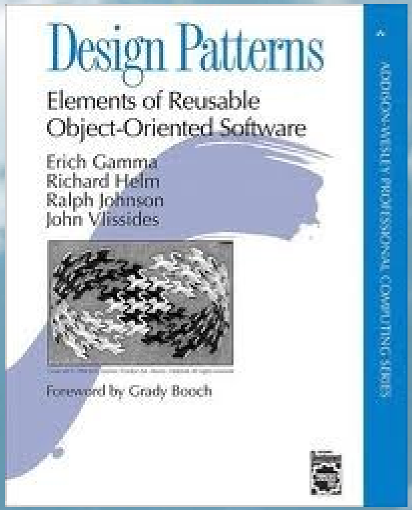
## Five patterns: Others

- Iterator
- Observer
- State
- Strategy
- Lazy loading
- Command
- Abstract factory
- Factory method
- Mediator
- Template method
- etc.

## Review and closing thoughts

- Object-oriented programming is just one paradigm
  - Pros and cons
- Many design principles for good OO design
- Design patterns help promote and incorporate OO principles

## Resources: Books



*Head First Design Patterns* by Elisabeth Freeman, Eric Freeman

*Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides

## Resources: Articles

- “Designing Reusable Classes” by Ralph E. Johnson and Brian Foote (1998)
- Robert C. Martin
  - “Open Closed Principle” (1996)
  - “Liskov Substitution Principle” (1996)
  - “Dependency Inversion Principle” (1996)
  - “Interface Segregation Principle” (1996)
  - “Single Responsibility Principle” (2002)
  - “Principles and Patterns” (2000)

## Notes

- Clouds OOo template by Jonty Pearce
- No rights reserved. (**CC0** where applicable.)