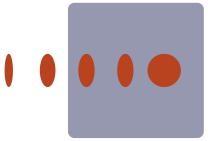
Aspect-Oriented Programming

William Candillon

{wcandillon@elv.telecom-lille1.eu}



TELECOMLille1

ECOLE D'INGENIEURS

Pop Quiz!


GUINNESS
DRAUGHT
GUINNESS
DRAUGHT
GUINNESS
DRAUGHT
GUINNESS
DRAUGHT

Problem

- Every call to *addItem()* are logged

```
$myOrder->addItem('Largo Winch', 3);
log('3 Largo Winch added');
```

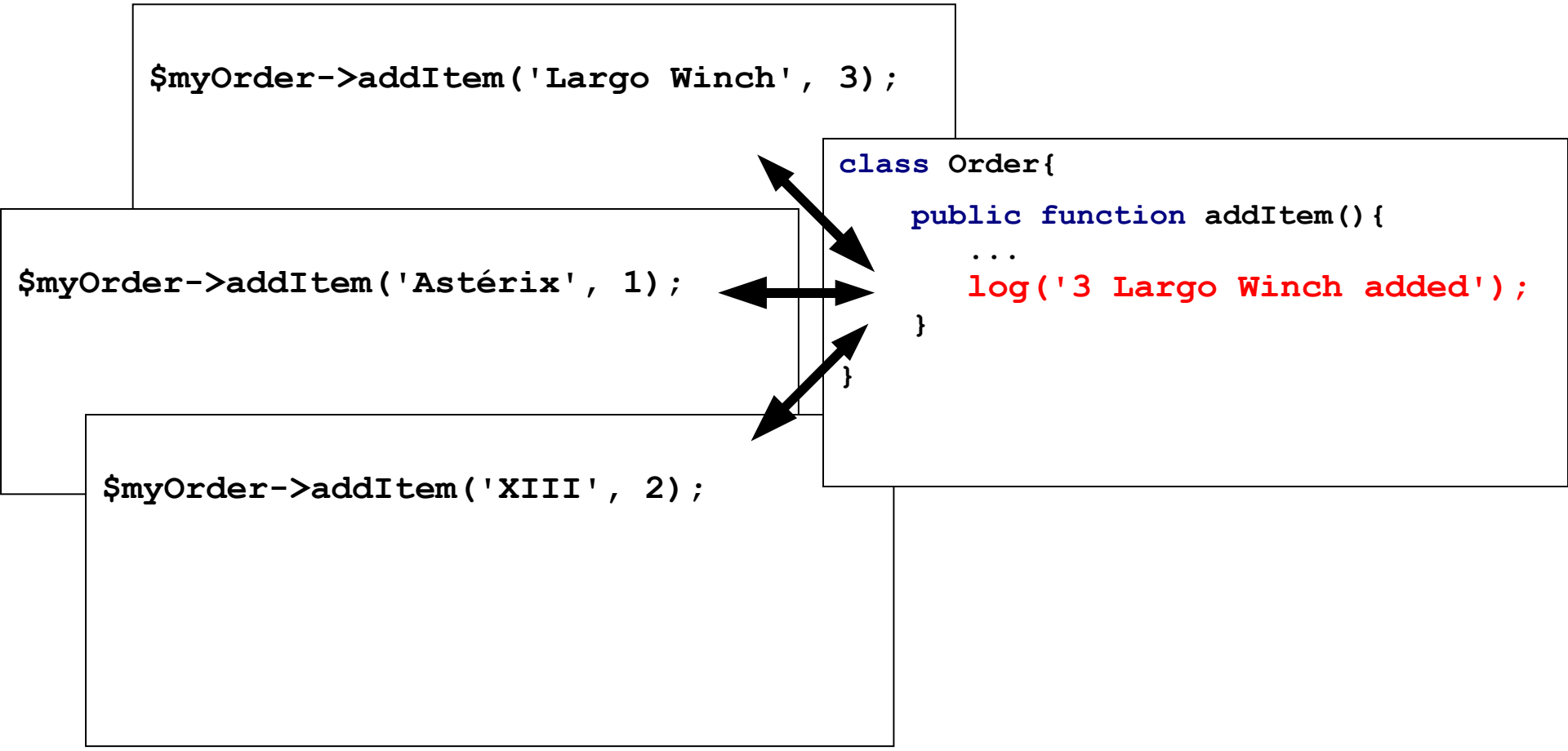
```
$myOrder->addItem('Astérix', 1);
log('1 Astérix added');
```

```
$myOrder->addItem('XIII', 2);
log('2 XIII added');
```

```
class Order{
    public function addItem(){
        ...
    }
}
```

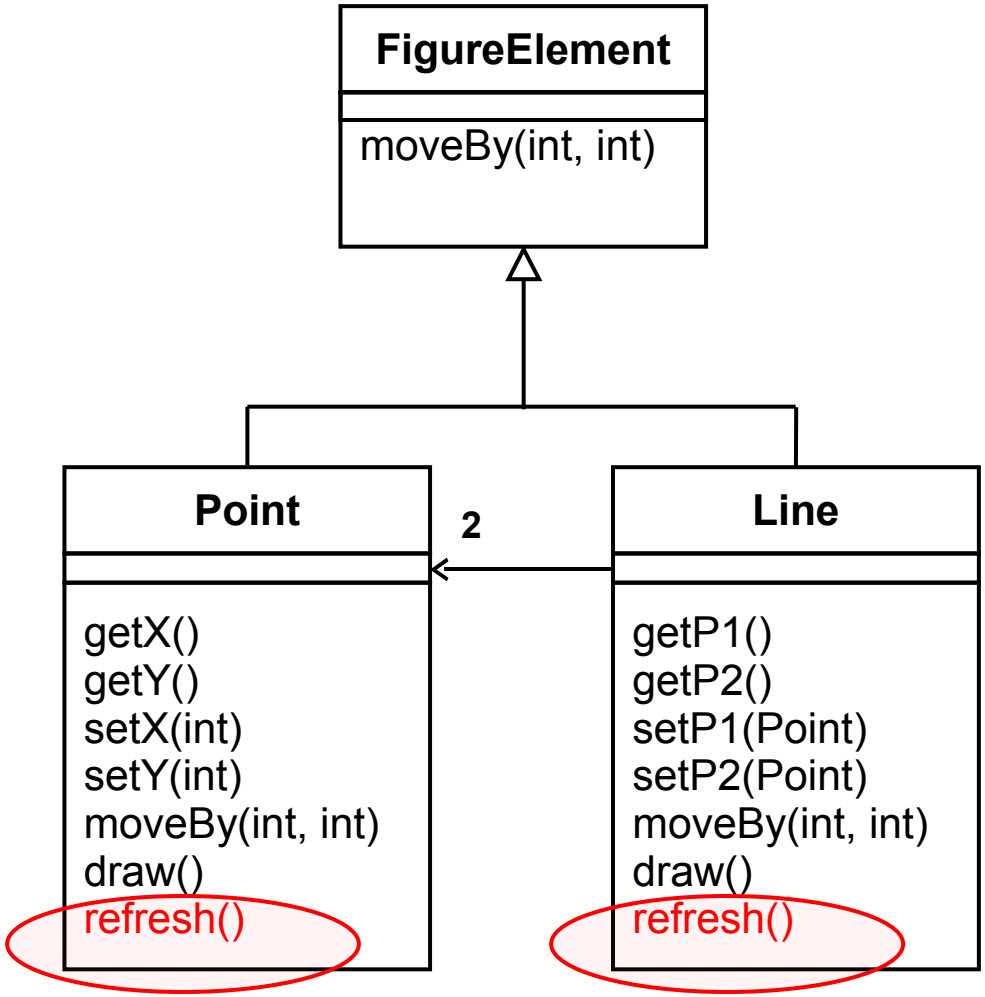
Solution

- Procedures can modularize this case



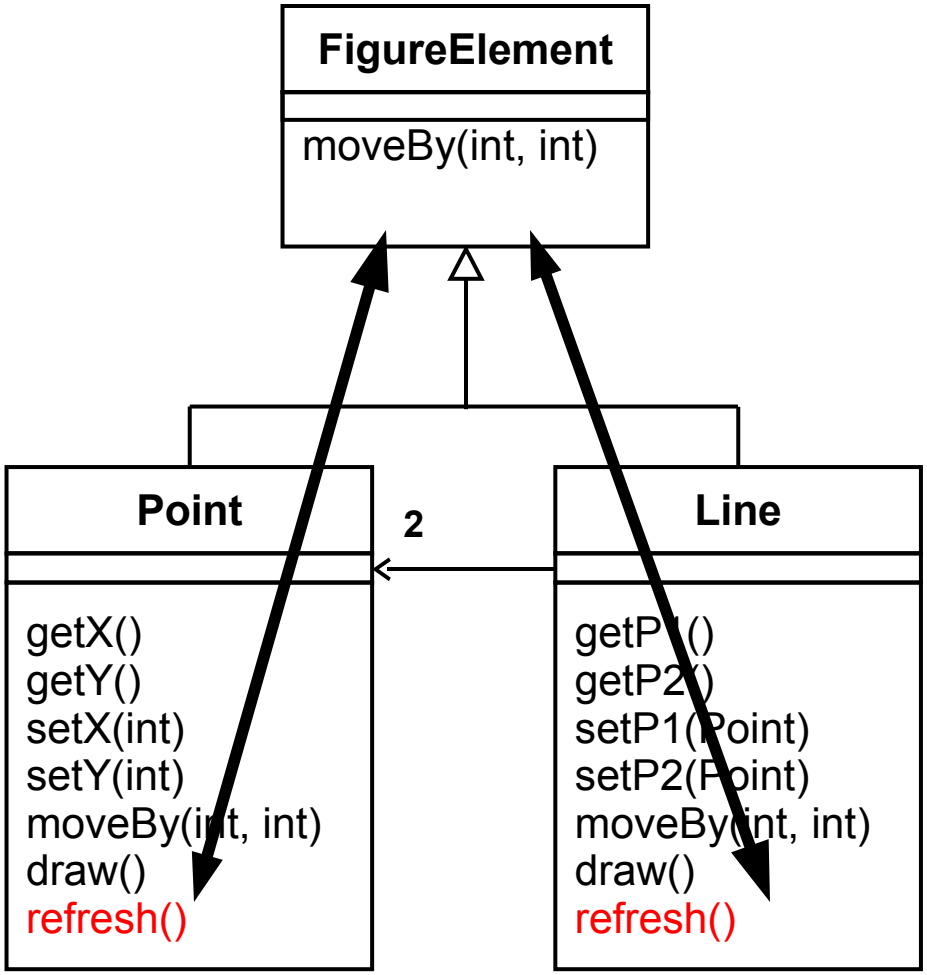
Problem

- All subclass have an identical method



Solution

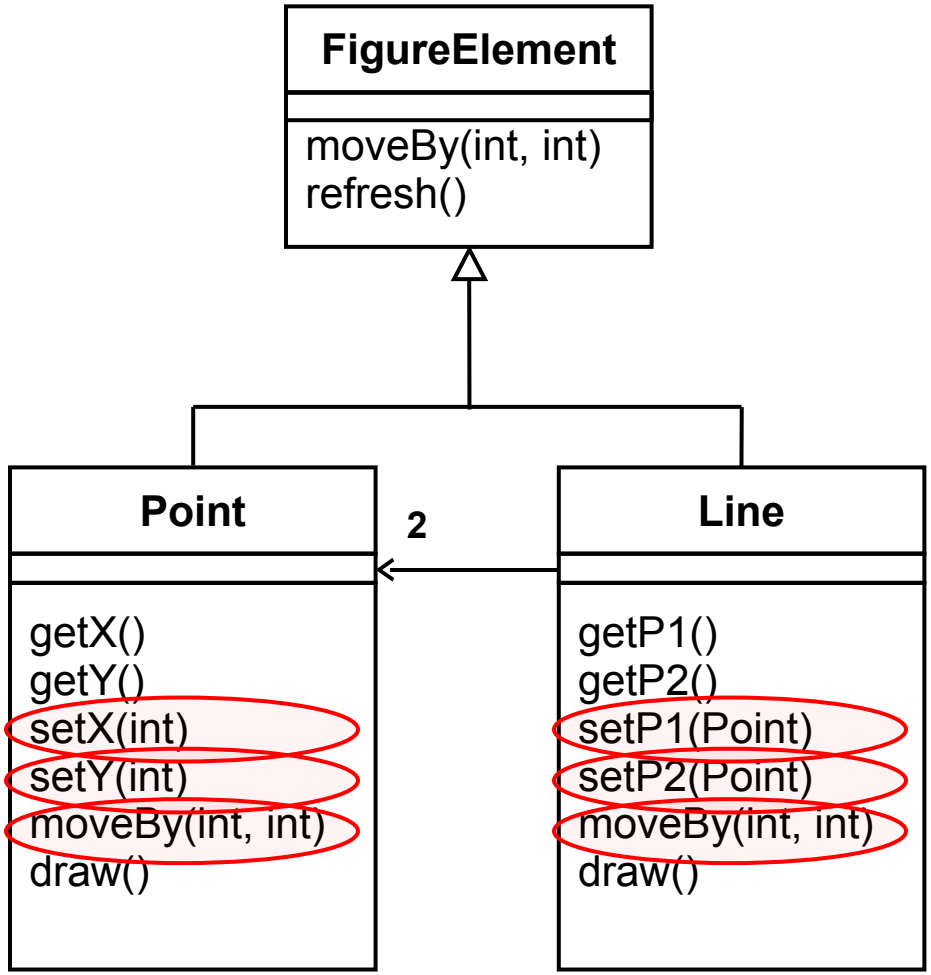
- Inheritance



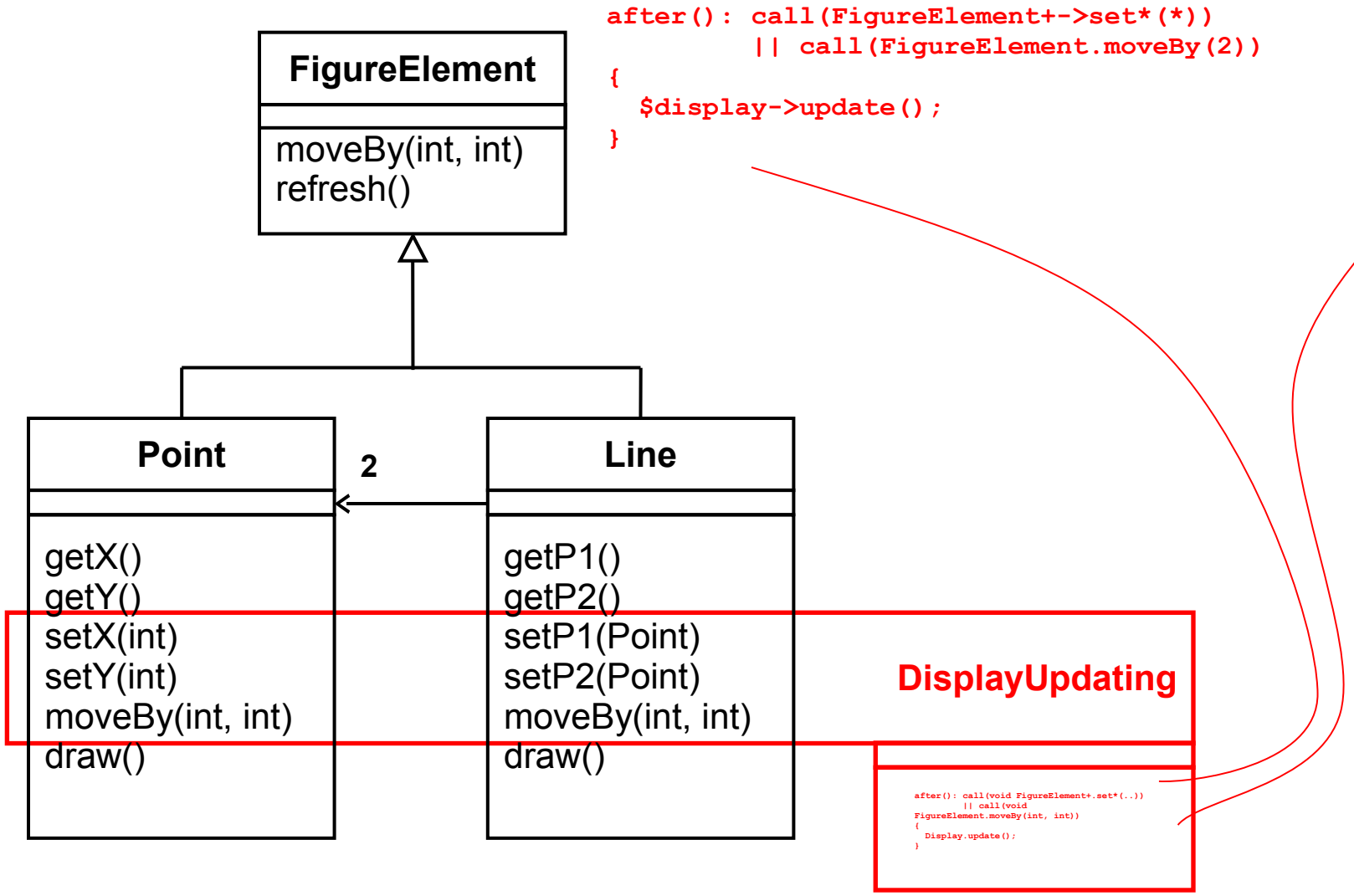
Problem

- These methods all end with a call to:

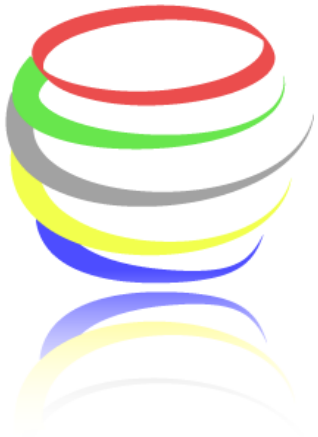
```
$display->update()
```



An aspect!

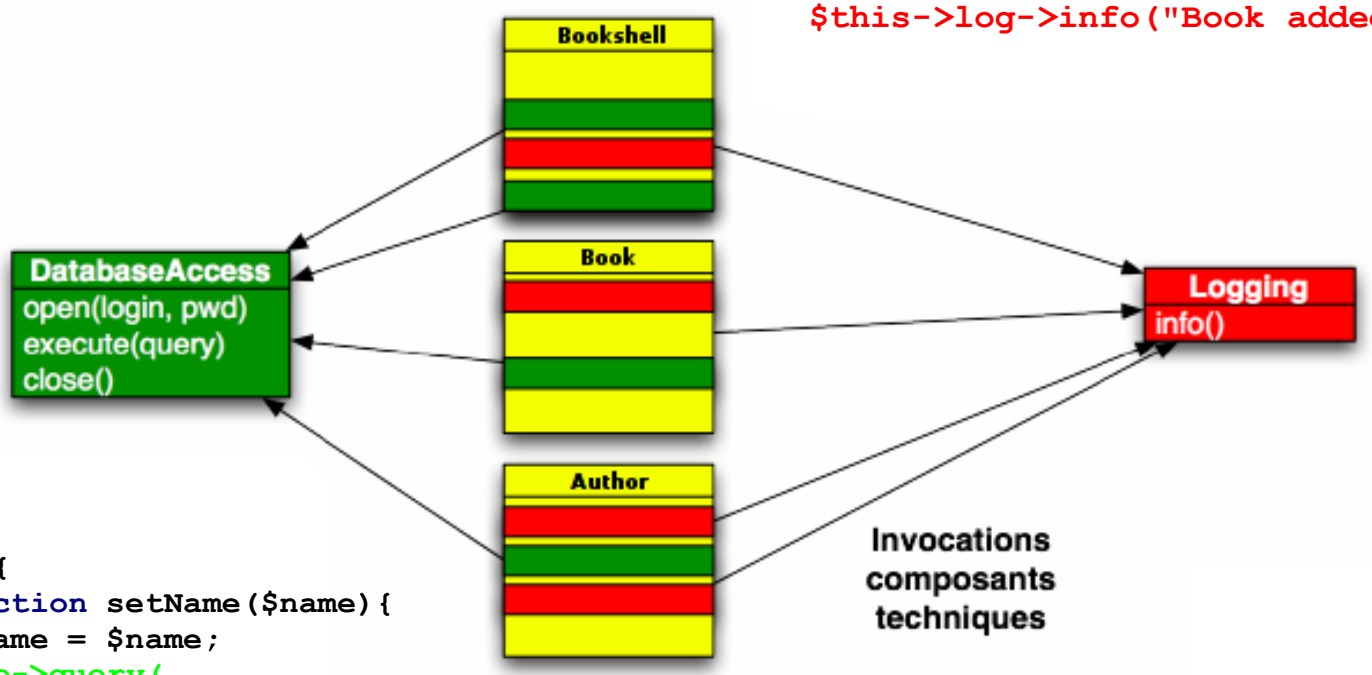

after(): call(FigureElement+->set*(*))
|| call(FigureElement.moveBy(2))
{
$display->update();
}
FigureElement
moveBy(int, int)
refresh()
Point
getX()
getY()
setX(int)
setY(int)
moveBy(int, int)
draw()
2
Line
getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)
draw()
DisplayUpdating
after(): call(void FigureElement+.set*(..))
|| call(void
FigureElement.moveBy(int, int))
{
Display.update();
}

Introduction to AOP



Technicals aspects in OOP

```php
class Bookshelf{
  public function addBook($id){
    $this->books[] = new Book($id);
    $this->log->info("Book added: %d", $id);
```



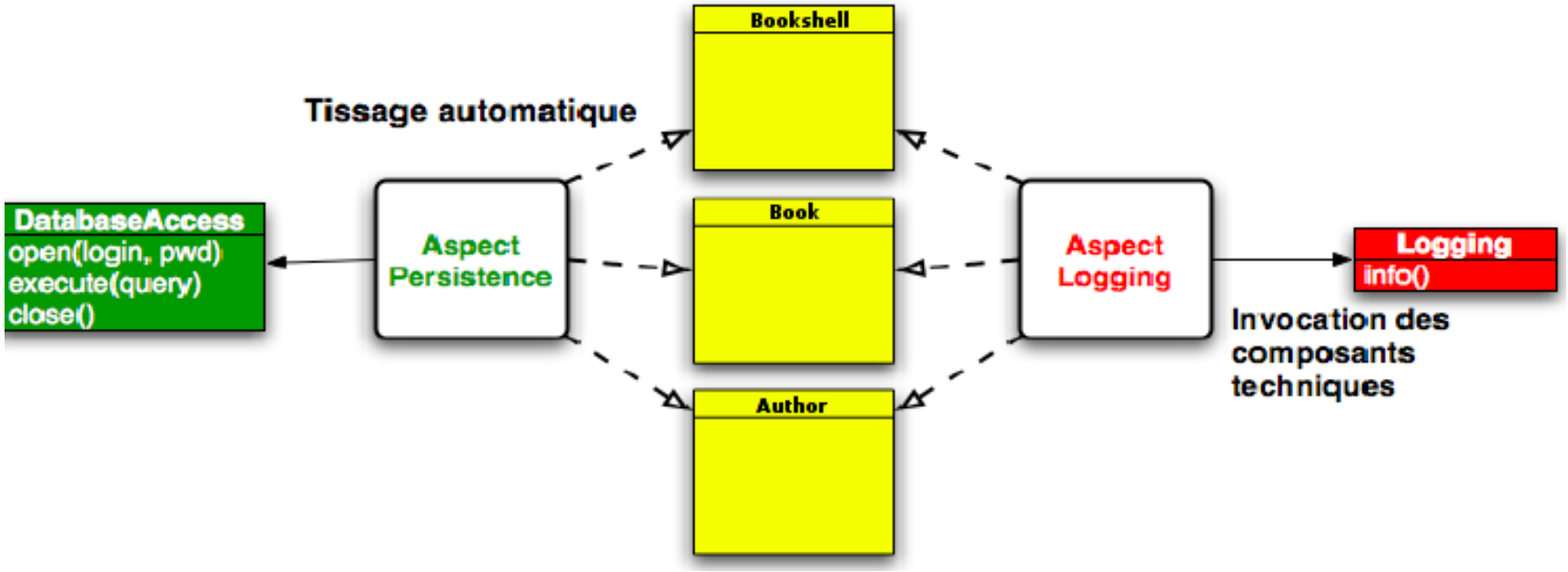

```php
class Author{
  public function setName($name){
    $this->name = $name;
    $this->db->query(
      "UPDATE author SET name = {$name}
        WHERE author_id = {$this->id}"
    );
```

Symptoms

- Inheritance can't modularize crosscutting concerns
- Code Tangling
- Code scattering
- Difficulties
  - Code comprehension
  - Code reusability
  - Code evolution

General idea

- Inversion of technicals dependencies



Aspect-Oriented Programming (1/2)

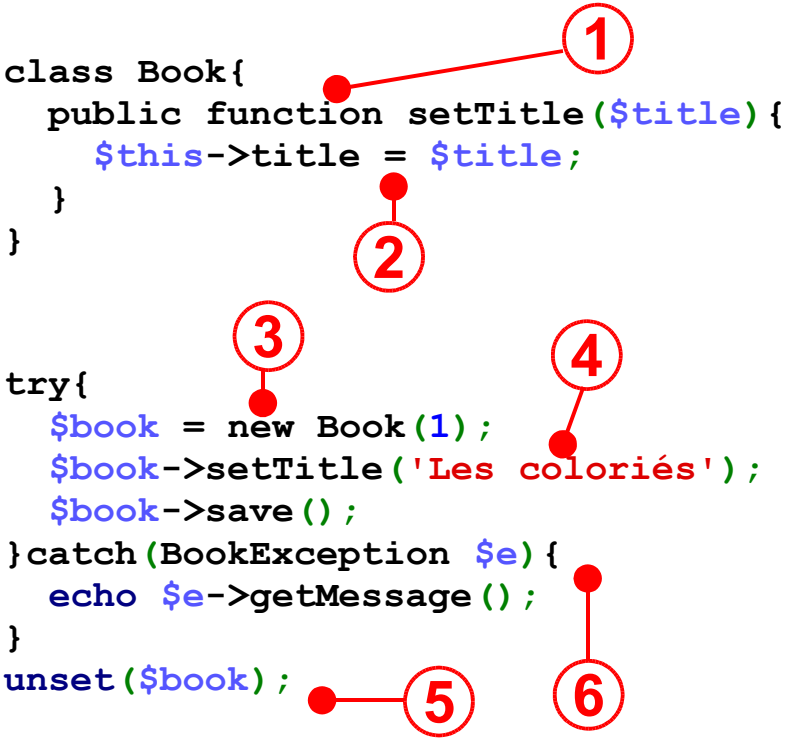
- A new programming paradigm...
  - Working with OOP
  - Split technicals concerns from the business logic
- ...defining mechanisms for
  - Writing aspects as a new software entity
  - Wrapping crosscutting concerns
  - Adding technicals concerns to business logic

Aspect-Oriented Programming (2/2)

- An active field of research and development: http://scholar.google.com/scholar?q=aop
- De facto standard: aspectJ (http://www.eclipse.org/aspectj)

Joinpoints

- Identified instructions in the program flow:
  - Method execution **(1)**
  - Attribute writing/reading **(2)**
  - Objects construction **(3)**
  - Method call **(4)**
  - Objects destruction **(5)**
  - Exception throwing **(6)**




```
class Book{
  public function setTitle($title){
    $this->title = $title;
  }
}

try{
  $book = new Book(1);
  $book->setTitle('Les coloriés');
  $book->save();
}catch(BookException $e){
  echo $e->getMessage();
}
unset($book);
```

- A pointcut is a logical association of joinpoints

Code advices

- Injected code in pointcuts
- 3 types:
  - Before
  - Around
  - After

```
before HelloWorld{
  echo "Before saying Hello World\n";
}

around HelloWorld{
  if($condition){
    proceed();
  }else{
    echo "\nCan't say Hello World\n";
  }
}

after HelloWorld{
  echo "After said Hello World\n";
}
```

- Aspect and joinpoint reflection to explore the context
  - Methods, arguments, targeted object etc...

```
around new(Foo(*)) || new(Bar(2)){
   $className = $thisJoinPoint->getClassName();
   if(!isset($thisAspect->instances[$className])){
       $thisAspect->instances[$className] = proceed();
   }
```

Inter-type declarations

- Inter-type attributes

```
private Bo*::$pearLog, Bo*::$debug = false;
```

- Inter-type constants

```
Log::PEAR_LOG_DEBUG = 7, Log::PEAR_LOG_ERR = 3;
```

- Inter-type methods

```
public function Bo*::setLog(Log $log){
  if($this->debug){
    $log->setMask(Log::PEAR_LOG_DEBUG);
  }else{
    $log->setMask(Log::PEAR_LOG_ERROR);
  }
  $this->pearLog = $log;
}
```

Other features

- Logicals operators:

```
poincut changeState:(exec(* *::add*(1)) || exec(* *::set*(1)))
                        && !exec(* *::addBook(1));
```

- Enum operator:

```
public function Foo,Bar::__clone(){
  trigger_error('Clone is not allowed.', E_USER_ERROR);
}
```

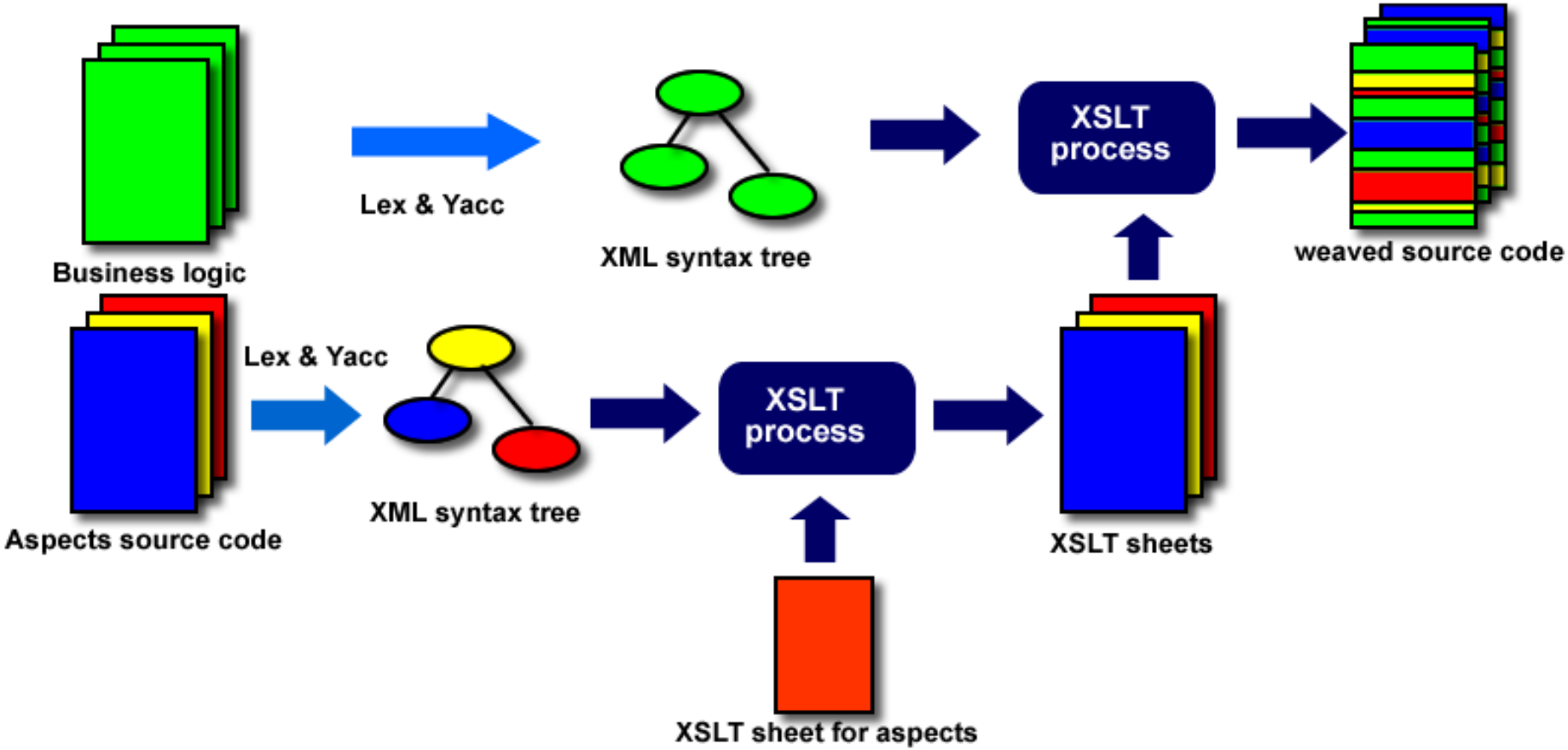
- New joinpoints

```
pointcut changeBook:exec(* *::set*(1)) && @return true;
```

- Aspect introspection


phpAspect
BETA

Weaving chain of phpAspect



A simple piece of PHP...

```php
<?php
class Order{

  private $items = array();
  private $amount = 0;

  public function addItem($reference, $quantity){
    $this->items[] = array($reference, $quantity);
    $this->amount += $quantity*Catalog::getPrice($reference);
  }

  public function getAmount(){ return $this->amount; }
}

class Catalog{
  private static $priceList = array('Largo Winch' => 9.31,
    'Astérix' => 8.46, 'XIII' => 8.70);

  public static function getPrice($reference){
    return self::$priceList[$reference];
  }
}

$myOrder = new Order;
$myOrder->addItem('Largo Winch', 2);
$myOrder->addItem('Astérix', 2);
$myOrder->addItem('Largo Winch', -6);
?>
```

- A client add products in the cart.
- Business logic without any technicals concerns.

The most ~~boring~~ common example

```php
<?php
aspect TraceOrder{
  pointcut logAddItem:exec(public Order::addItem(2));
  pointcut logTotalAmount:call(Order->addItem(2));

  after logAddItem{
    printf("%d %s added to the cart\n", $quantity,
$reference);
  }

  after logTotalAmount{
    printf("Total amount of the cart : %.2f €\n",
      $thisJoinPoint->getObject()->getAmount());
    }
  }
?>
```

- After weaving

```php
$myOrder->catalog = new Catalog;
$myOrder->addItem('Largo Winch', 2);
printf("Montant total de la commande : %.2f €\n",
        $myOrder->getAmount());
$myOrder->addItem('Astérix', 1);
printf("Montant total de la commande : %.2f €\n",
        $myOrder->getAmount());
```

- Something went wrong...

```
2 Largo Winch added to the cart
Total amount of the cart : 18.62 €
1 Astérix added to the cart
Total amount of the cart : 27.08 €
-6 Largo Winch added to the cart
Total amount of the cart : -28.78 €
```

A security aspect

```
<?php
aspect Security{
  pointcut logAddItem:exec(public Order::addItem(2));

  before logAddItem{
    if(!Catalog::getPrice($reference) ||
         (float)$quantity < 0){
      echo "Wrong parameters";
      return false;
    }
  }
?>
```

- Make a filter on the customer input
- Protection against cross scripting injection
- Result

```
2 Largo Winch added to the cart
Total amount of the cart : 18.62 €
1 Astérix added to the cart
Total amount of the cart : 27.08 €
Wrong parameters
Total amount of the cart : 27.08 €
```

Web-specific crosscutting-concerns

- PHP contains a lot of highly sensitive information that is readable and writable at any level in any context:
  - $_GET, $_POST, $_COOKIE, $_SERVER, $_SESSION

- We need a web-specific joinpoint to ensure matters such as security

```
pointcut XssProtect: get($_POST[*]) ||  set($_POST[*]);
```

Web-specific crosscutting-concerns

- PHP has a specific execution model.
- By default, one singleton of each aspect is created for one request.
- You can specify one singleton per session.
- Or one instance to follow an url pattern.

On our logging example

- `Aspect Logging from('*/checkout/'){ //... }`

- When a visitor arrives from the mysite.com/checkout/ url, the runtime get the same aspect instance than on the previous request.

- Fits with REST webservices.
- Aspect user experience.

Generic aspects

- Using XML files to parameterize aspects.
- Taking advantage of the implementation with XSLT.

```
aspect Singleton{

   public $instances = array();

   pointcut singleton(<singleton.class> $s):new($s(*));

   around(): singleton{
       $className = $thisJoinPoint->getClassName();
       if(!isset($this->instances[$className])){
           $this->instances[$className] = proceed();
       }
       return $this->instances[$className];
   }
}
```

```
<singleton>
  <class>Foo</class>
  <class>Bar</class>
</singleton>
```

Another example

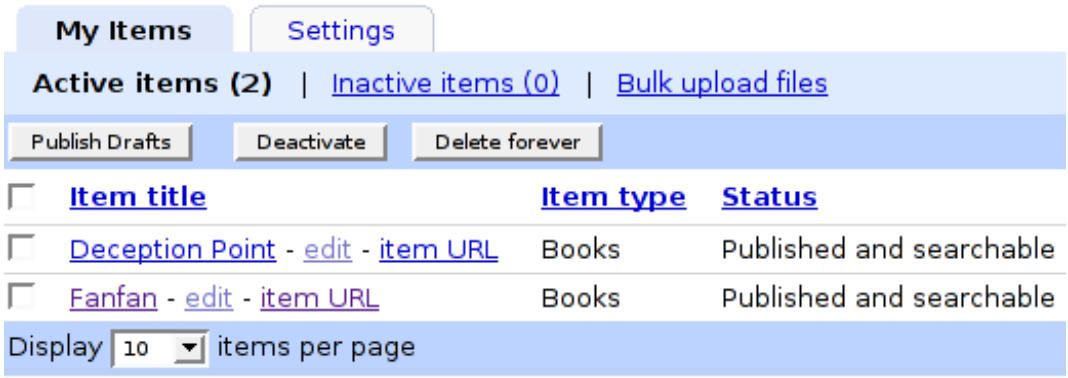
```
class Bookshelf{
  private $books = array();

  public function addBook(Book $book){
    $this->books[$book->getTitle()]
                        = $book;
  }

  public function getBook($title){
   return $this->books[$title];
  }

  public function deleteBook($title){
    unset($this->books[$title]);
  }
}
class Book{
 private $title;
 private $author;
 //Setters and getters...
}

$book = new Book;
$book->setTitle('Fanfan');
$book->setAuthor('Alexandre Jardin');

$myBookShelf = new BookShelf;
$myBookShelf->addBook($book);
```

- We need persistence:
  - Google Base would be cool
  - But we also want to be able to switch easily on something else (SQL, XML, etc)
  - Without breaking code design and modularity
- Adding a data layer to get the application offline (Google gears)

A new aspect of persistence

```
aspect GoogleBaseServices{

 after($book) exec(public BookShelf::addBook(*)){
  $thisAspect->addEntry($book->toXML());
 }

 after($book) exec(BookShelf->deleteBook(*)){
  $thisAspect->deleteEntry(
                    $this->getBooks($title)
                       ->toXML());
 }

  public function BookShelf::toXML(){
   /*XML Representation of
     a Google Base Book item */
   $xml = "<entry>
              <title>{$this->title}</title>
              ...
           </entry>";
    return $xml;
  }

  /* HTTP requests to google base */
}
```

- Publish the bookshelf on Google Base without modifying the previous source code
- All the persistent code is factorized in one software entity
- Result (after weaving):



APDT, the eclipse plugin

- Google Summer of Code project for this year.
- Working with PDT (eclipse.org/pdt).
- Nothing as ambitious as AJDT.
- Just (for now):
  - Syntax support and highlighting.
  - Integrating the phpAspect builder.
  - Running the weaved application.

- Due to the dynamic nature of PHP other features are really challenging to implement

Thanks for your attention