Magento Live

UK | 2016

Key Insights into Development Design Patterns for Magento 2

MagentoLive
UK | 2016

Max Pronko

CTO @ GiftsDirect, TheIrishStore

Today's Presentation Includes

- Design Patterns and Magento 2
- Aspect Oriented Programming
- Questions and Answers


2
Design Patterns

“

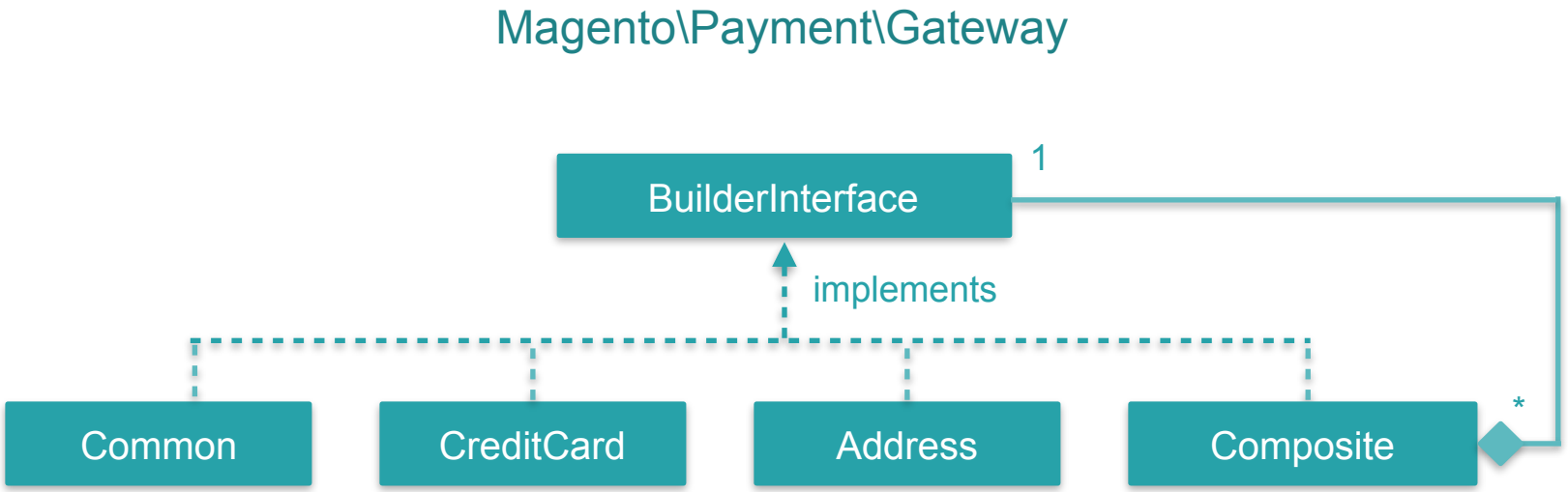
describes a problem which occurs over and over again, and then describes the core of the solution to that problem, without ever doing it the same way twice

- Christopher Alexander

Composite Pattern

Composite pattern is used to treat a group of objects in similar way as a single object uniformly

Composite Diagram



BuilderComposite Example

```
class BuilderComposite implements BuilderInterface
{
    /** @var BuilderInterface[] */
    private $builders;

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

    public function build(array $buildSubject) {
        $result = [];
        foreach ($this->builders as $builder) {
            $result = array_merge_recursive($result, $builder->build($buildSubject));
        }

        return $result;
    }
}
```

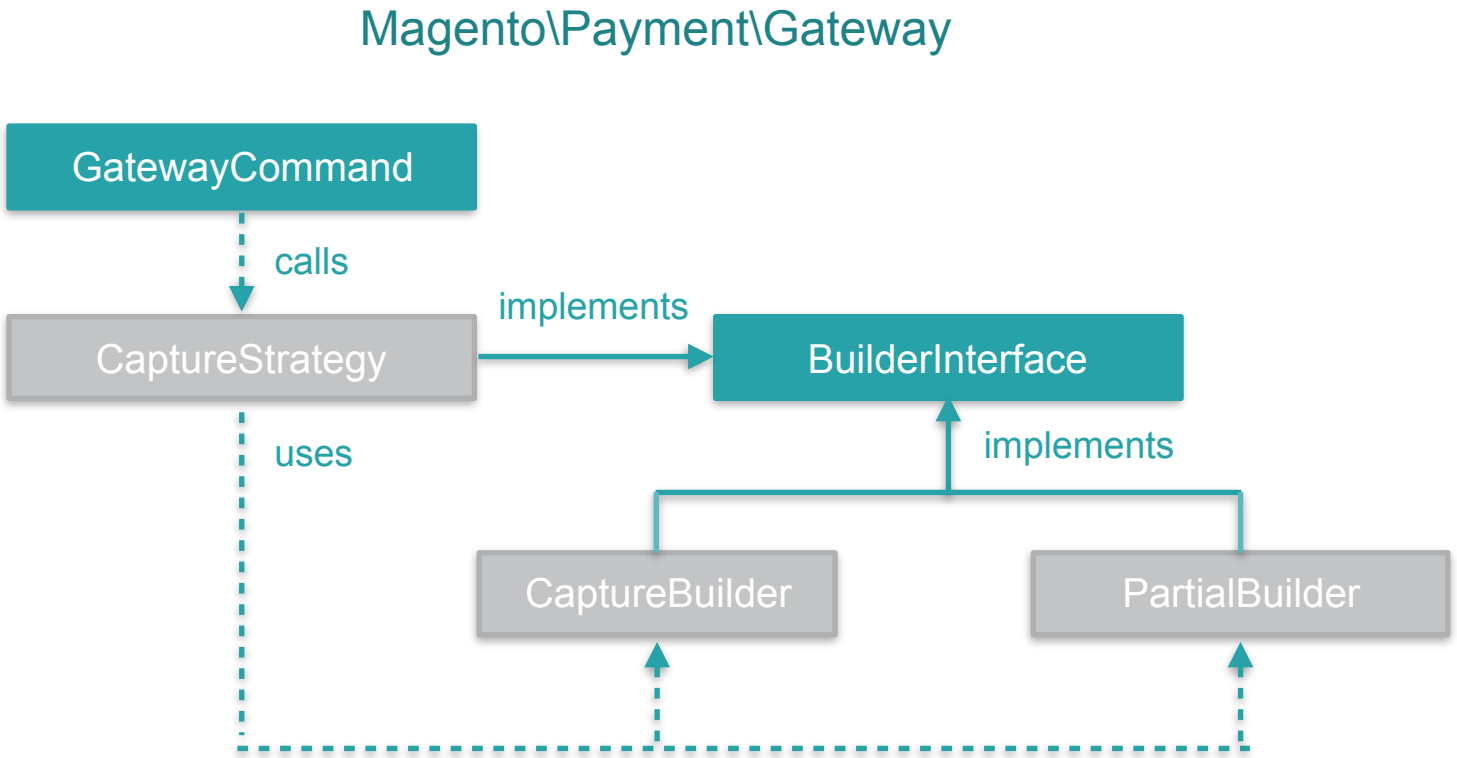
CompositeBuilder Declaration

```
<config>
   <virtualType name="CaptureBuilderComposite" type="Magento\Payment\Gateway\Request\BuilderComposite">
      <arguments>
         <argument name="builders" xsi:type="array">
            <item name="common" xsi:type="object">Pronko\Payment\Gateway\CommonBuilder</item>
            <item name="credit_card" xsi:type="object">Pronko\Payment\Gateway\CreditCardBuilder</item>
            <item name="address" xsi:type="object">Pronko\Payment\Gateway\AddressBuilder</item>
         </argument>
      </arguments>
   </virtualType>
</config>
```

Strategy Pattern

Strategy lets the algorithm vary independently from the clients that use it

Strategy Pattern Diagram



Strategy Pattern

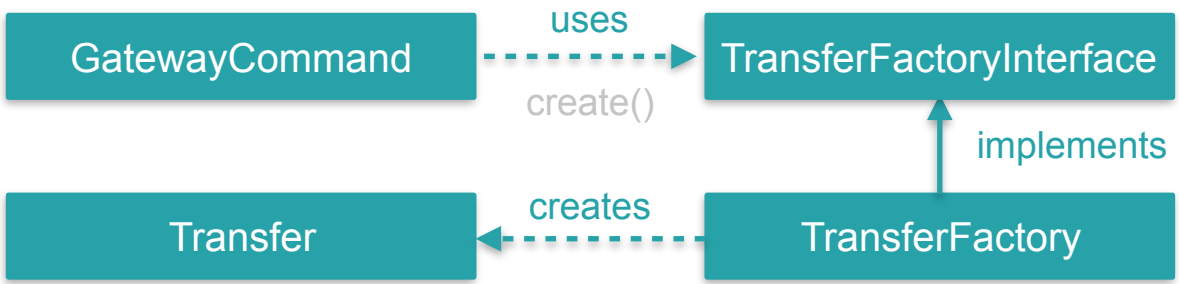
```
class CaptureStrategy implements BuilderInterface {
    /** @var BuilderInterface */
    protected $partial;
    /** @var  BuilderInterface */
    protected $capture;

    public function build(array $buildSubject) {
        $condition = //set condition

        if ($condition) {
            return $this->partial->build($buildSubject);
        }

        return $this->capture->build($buildSubject);
    }
}
```

Factory Method Pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

Factory Method Pattern Diagram

Magento\Payment\Gateway\Command



Transfer Factory Example

```
namespace Magento\Braintree\Gateway\Http;

class TransferFactory implements TransferFactoryInterface
{
    private $transferBuilder;

    public function __construct(TransferBuilder $transferBuilder) {
        $this->transferBuilder = $transferBuilder;
    }

    public function create(array $request) {
        return $this->transferBuilder
            ->setBody($request)
            ->build();
    }
}
```

Transfer Factory Example

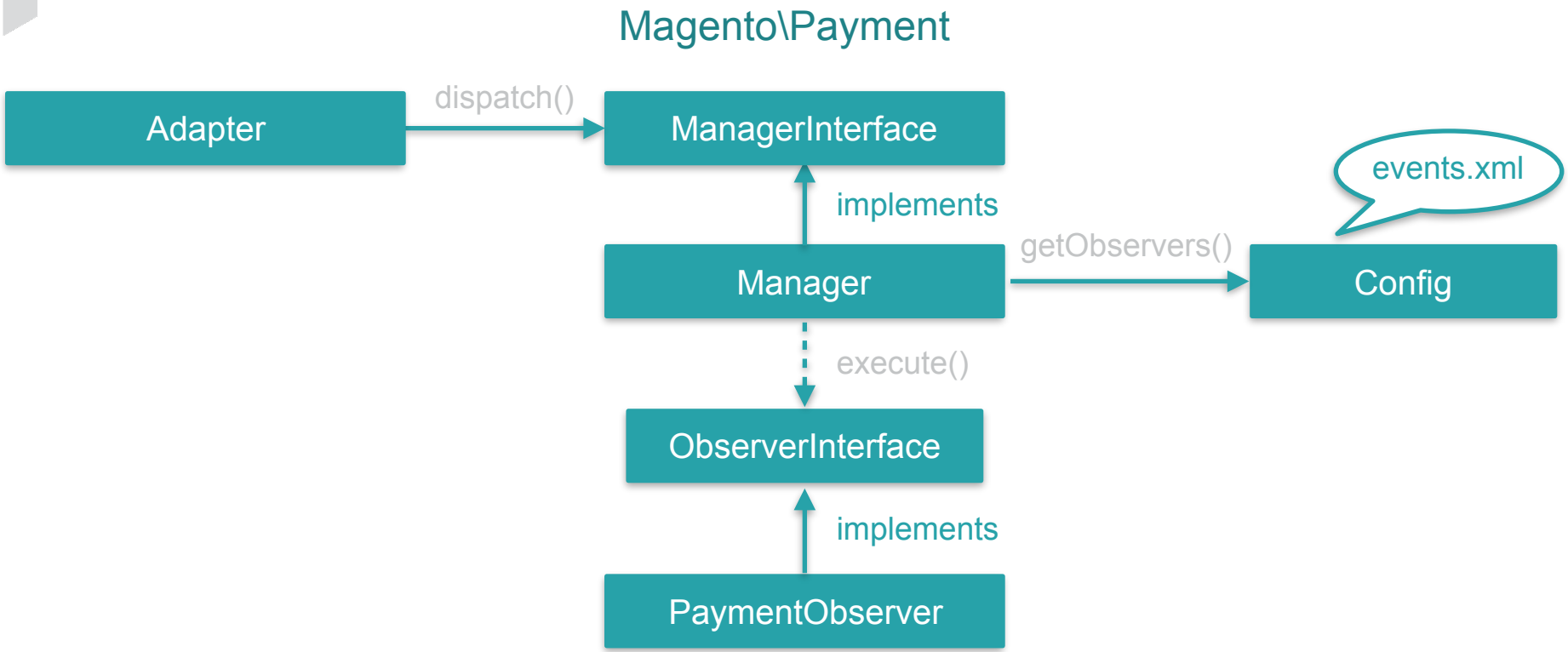
```php
namespace Magento\Braintree\Gateway\Http;

class TransferFactory implements TransferFactoryInterface
{
    private $transfer

    public function _
        $this->transf
    }

    public function c
        return $this-
            ->setBody(
            ->build();
    }
}
```

```php
class GatewayCommand implements CommandInterface {
    public function execute(array $commandSubject) {
        $transferO = $this->transferFactory->create(
            $this->requestBuilder->build($commandSubject)
        );

        $response = $this->client->placeRequest($transferO)
        // … code
    }
}
```

Observer Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Observer Pattern Diagram



Dispatching an event

```
use Magento\Framework\Event\ManagerInterface;

class Adapter implements MethodInterface {
    /** @var  ManagerInterface */
    protected $eventManager;

    public function assignData(\Magento\Framework\DataObject $data) {
        $this->eventManager->dispatch(
            'payment_method_assign_data_' . $this->getCode(),
            [
                AbstractDataAssignObserver::METHOD_CODE => $this,
                AbstractDataAssignObserver::MODEL_CODE => $this->getInfoInstance(),
                AbstractDataAssignObserver::DATA_CODE => $data
            ]
        );

        return $this;
    }
}
```

Observer Configuration

```
<config>
   <event name=“payment_method_assign_data_custom">
      <observer name="custom_gateway_data_assign" instance="Pronko\Payment\Observer\DataAssignObserver" />
   </event>
</config>
```

Observer Implementation

```
namespace Pronko\Payment\Observer;

use Magento\Framework\Event\Observer;
use Magento\Payment\Observer\AbstractDataAssignObserver;

class DataAssignObserver extends AbstractDataAssignObserver
{
    public function execute(Observer $observer)
    {
        $data = $this->readDataArgument($observer);
        $additionalData = $data->getData(PaymentInterface::KEY_ADDITIONAL_DATA);
        $payment = $observer->getPaymentModel();

        $payment->setCcLast4(substr($additionalData->getData('cc_number'), -4));
    }
}
```

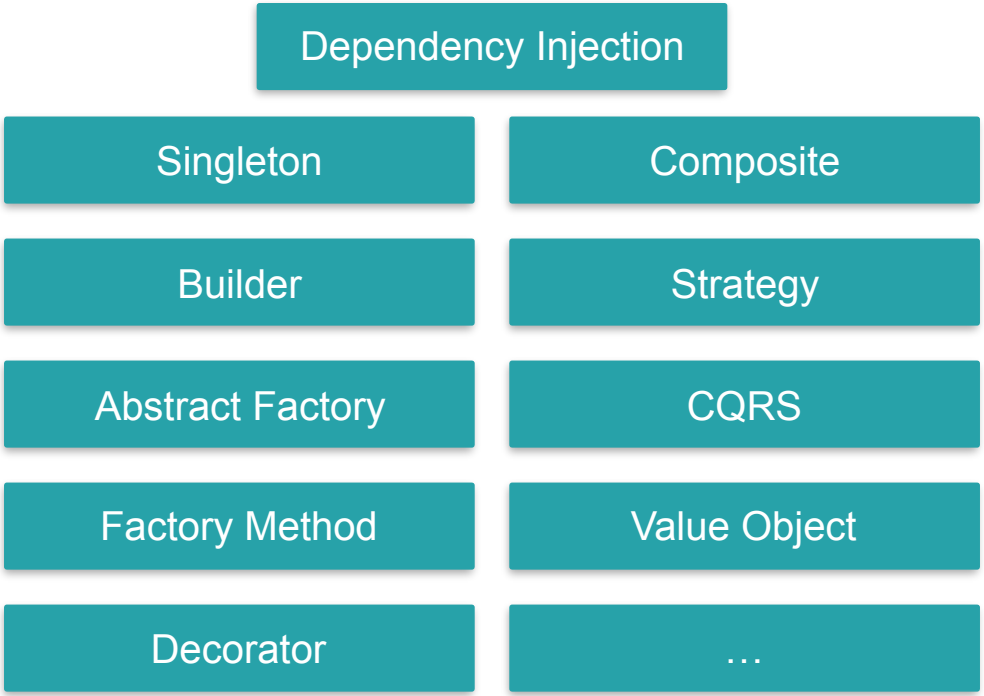
Object Manager

Typical implementation of Dependency Manager. Enables Inversion of Control support for Magento 2

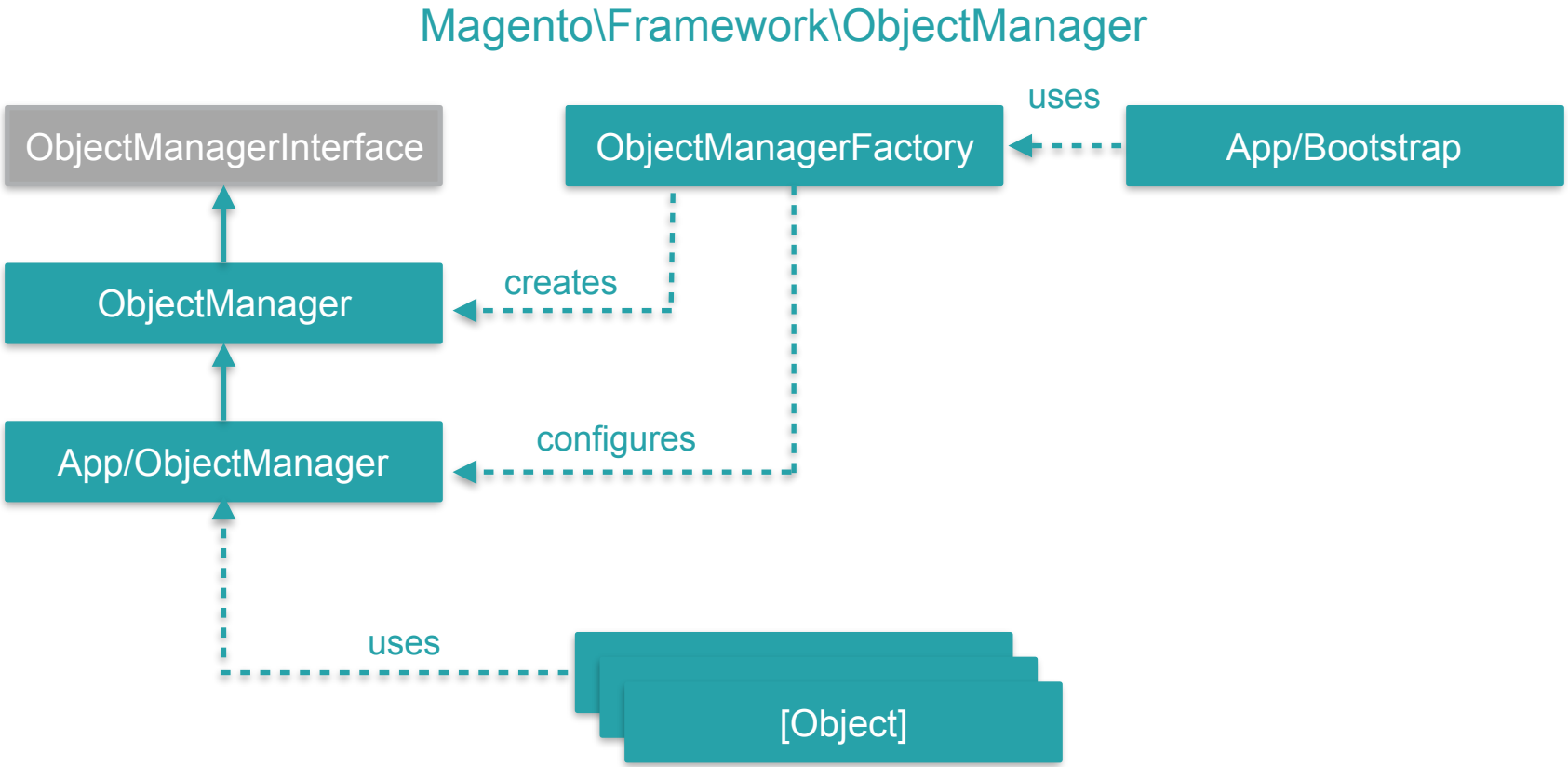
Object Manager

- Dependency Manager
- Instantiates classes
- Creates objects
- Provides shared objects pool
- Supports lazy Initialisation
- __construct() method injection

Object Manager Implementation



Object Manager Diagram



Object Manager Usage

Good

- Factory
- Builder
- Proxy
- Application
- generated classes

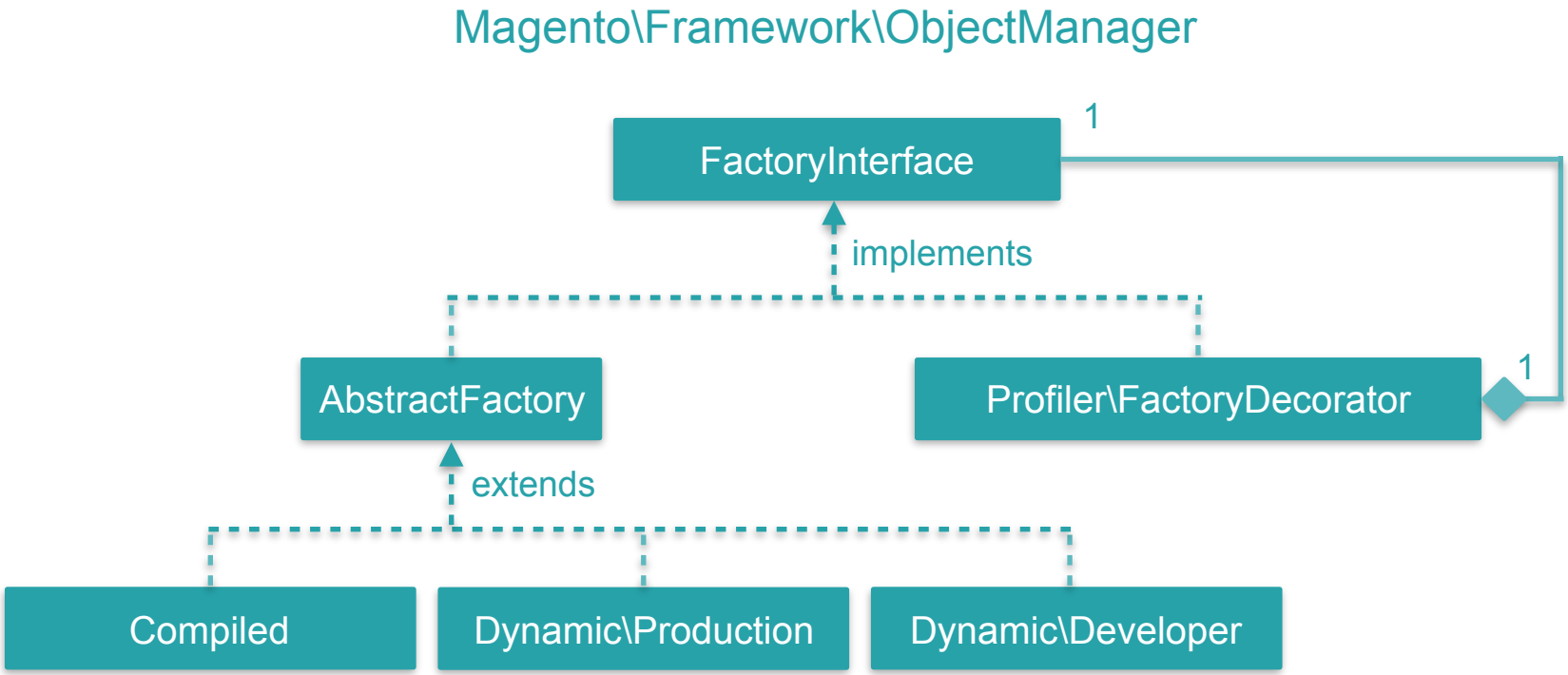
Bad

- Data Objects
- Business Objects
- Action Controllers
- Mage::getModel like calls
- Blocks

Decorator Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

Decorator Pattern Diagram



Decorator: Profiler Factory

```
namespace Magento\Framework\App\ObjectManager\Environment;

abstract class AbstractEnvironment implements EnvironmentInterface {
    protected function decorate($arguments){
        if (isset($arguments['MAGE_PROFILER']) && $arguments['MAGE_PROFILER'] == 2) {
            $this->factory = new FactoryDecorator(
                $this->factory,
                Log::getInstance()
            );
        }
    }
}
```

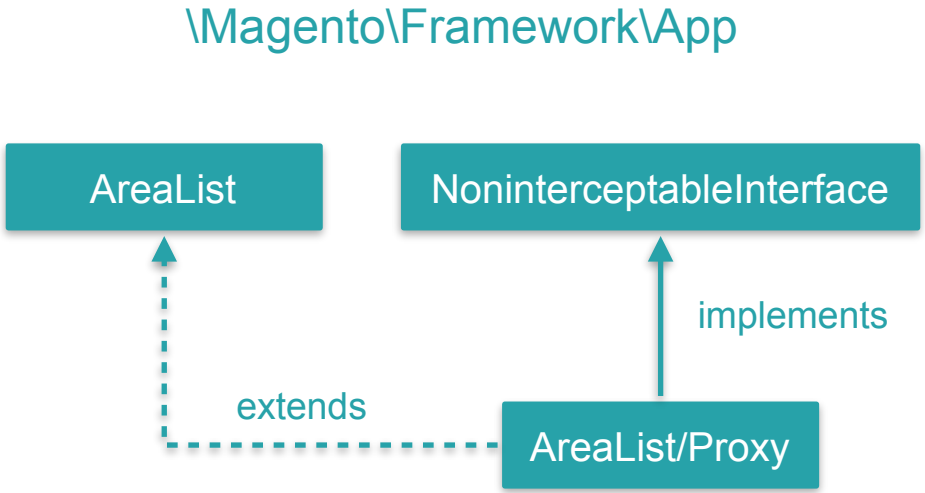
Decorator: Profiler Factory

```
namespace Magento\Framework\App\ObjectManager\Environment;

class DecoratorFactory implements FactoryInterface {}

class CompiledFactory implements FactoryInterface {}
```

Proxy Pattern

A proxy is a class functioning as an interface to something else. It is used for resource consuming objects

Proxy Pattern Diagram



Proxy Pattern Declaration

File: app/etc/di.xml

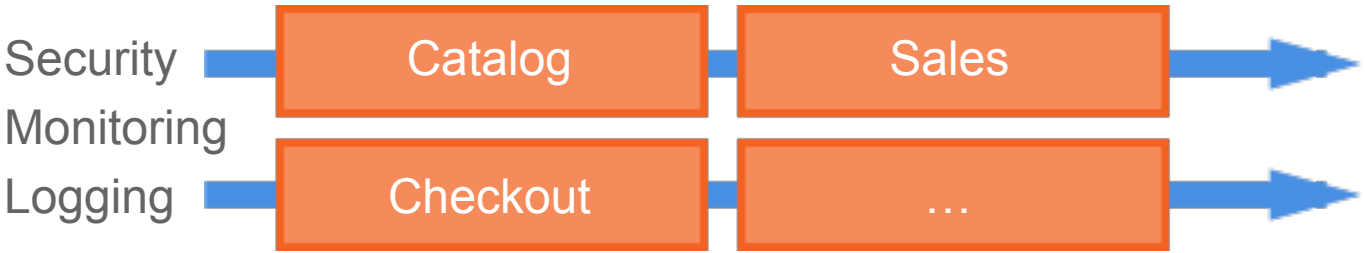
```
<config>
    <type name="Magento\Framework\Config\Scope">
        <arguments>
            <argument name="areaList" xsi:type="object">Magento\Framework\App\AreaList\Proxy</argument>
        </arguments>
    </type>
</config>
```


2
Aspect Oriented Programming

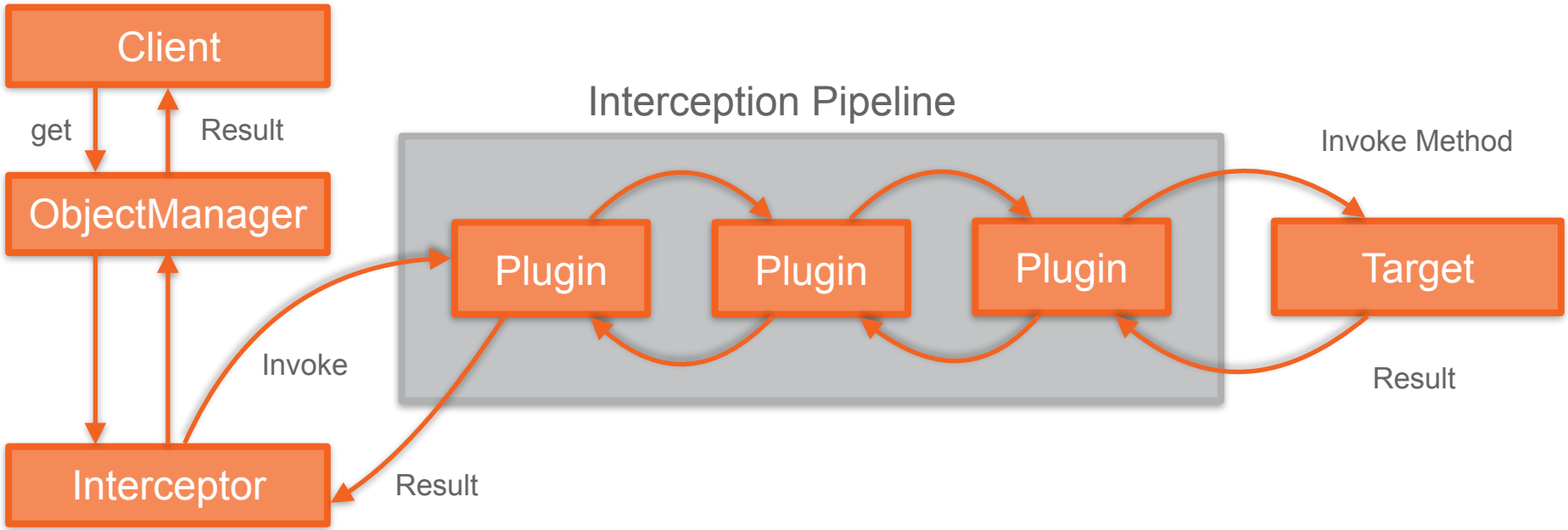
Aspect Oriented Programming

a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. The goal is to achieve loose coupling

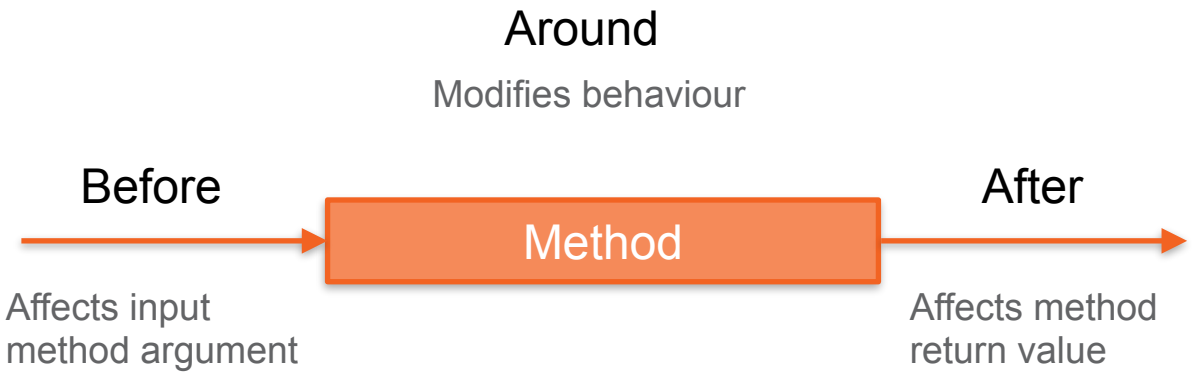
AOP Cross Cutting Concerns



Magento AOP Example



Types of a Plugin



Check Module Status Plugin

```
namespace Magento\Framework\Module\Plugin;

class DbStatusValidator
{
    public function aroundDispatch(
        \Magento\Framework\App\FrontController $subject,
        \Closure $proceed,
        \Magento\Framework\App\RequestInterface $request
    ) {
        if (!$this->cache->load('db_is_up_to_date')) {
            $errors = $this->dbVersionInfo->getDbVersionErrors();
            if ($errors) {
                // throw exception
            } else {
                $this->cache->save('true', 'db_is_up_to_date');
            }
        }
        return $proceed($request);
    }
}
```

Invalidate Cache Plugin

```
namespace Magento\WebapiSecurity\Model\Plugin;

class CacheInvalidator
{
    public function afterAfterSave(
        \Magento\Framework\App\Config\Value $subject,
        \Magento\Framework\App\Config\Value $result
    ) {
        if (\\condition) {
            $this->cacheTypeList->invalidate(\Magento\Webapi\Model\Cache\Type\Webapi::TYPE_IDENTIFIER);
        }

        return $result;
    }
}
```

Password Security Check Plugin

```
namespace Magento\Security\Model\Plugin;

class AccountManagement
{
    public function beforeInitiatePasswordReset(
        AccountManagementOriginal $accountManagement,
        $email,
        $template,
        $websiteId = null
    ) {
        $this->securityManager->performSecurityCheck(
            PasswordResetRequestEvent::CUSTOMER_PASSWORD_RESET_REQUEST,
            $email
        );
        return [$email, $template, $websiteId];
    }
}
```

Declaring Plugin

File: Magento\Security\etc\di.xml

```
<config>
    <type name="Magento\Customer\Model\AccountManagement">
        <plugin name="security_check_customer_password_reset_attempt" type="Magento\Security\Model\Plugin
\AccountManagement" />
    </type>
</config>
```

Interception

Benefits

- Public methods
- Auto-generated Decorators
- Flexible approach
- NoninterceptableInterface

Limitations

- Final methods / classes
- Static class and __construct() methods
- Virtual types

Summary

- Solve problems using Design Patterns
- Don’t overuse Design Patterns
- Build your code on abstractions (interfaces)
- Look inside Magento 2 for good practices

Thank you

Ask Me Anything

www.maxpronko.com

max_pronko