Architecture Design Patterns for PHP

Mark Niebergall

🤔 Question

- How do you run a fast marathon?

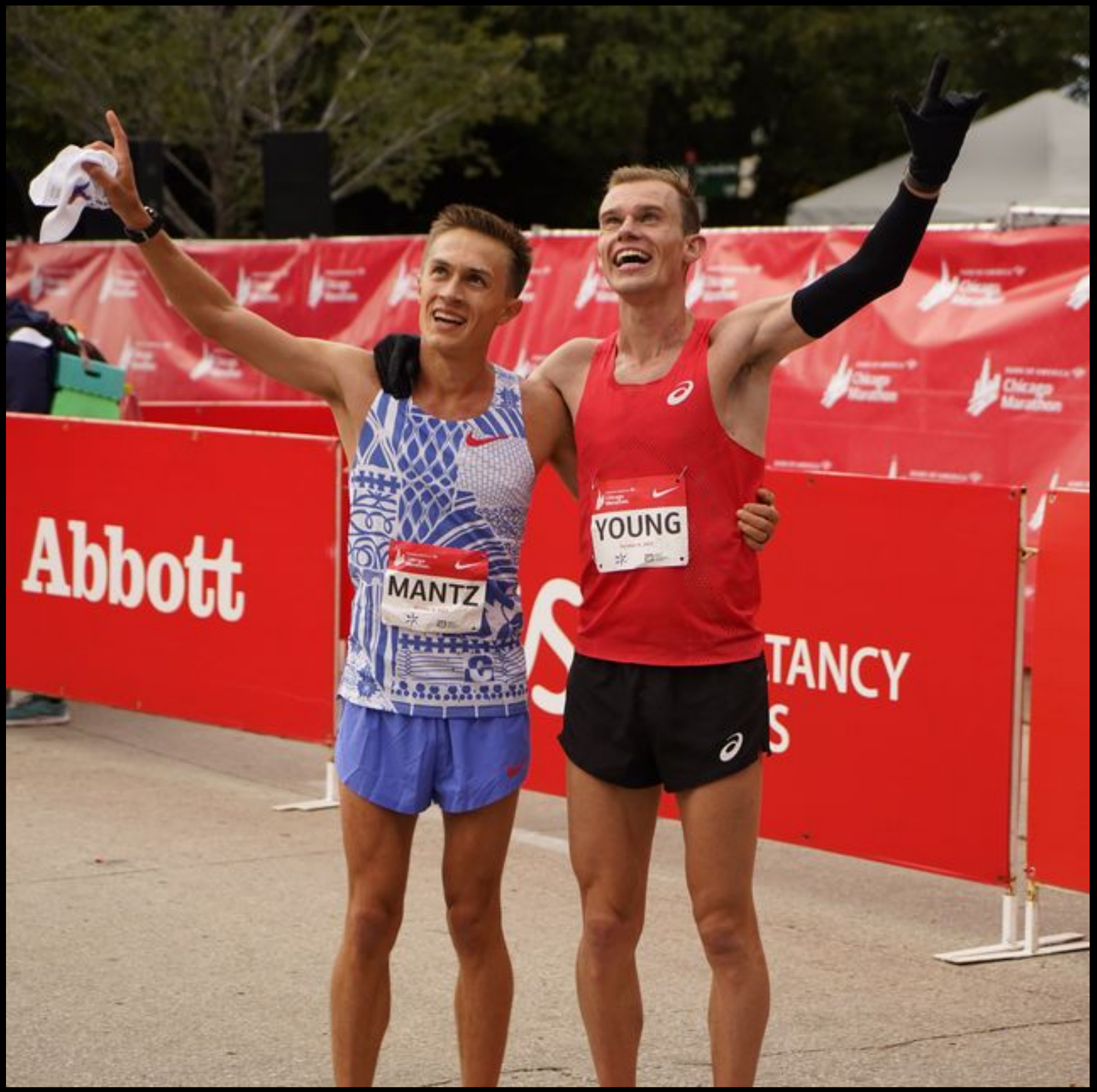


Photo by Derek Call

🤔 Question

- How do you run a fast marathon?
  - Established training run patterns
    - Speed Work
    - Tempo Runs
    - Long Runs
    - Recovery Runs
  - Strength Training
  - Diet, fueling, hydration
  - Race kit
  - Tapering pre-race
  - Mental and physical endurance

🤔 Question

- What code architecture design pattern should I use for this problem?

✅ Objective

- Familiar with pattern terminology
- Identify useful patterns for problems
- Start using patterns in your projects

📋 Overview

- Pattern Classifications
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
  - Other Patterns
- Benefits

Pattern Classifications

📂 Pattern Classifications

- Creational
- Structural
- Behavioral
- Concurrency

🏭 Creational Patterns

🏭 Creational Patterns

- Benefits
  - Isolates new-ing up classes
  - Testable code with mocks

🏭 Creational Patterns

- Patterns
  - Dependency Injection
  - Factory
  - Singleton

🏭 Creational Patterns

- Dependency Injection
  - Key design pattern
  - Inject things instead of creating them
  - Testability

🏭 Creational Patterns

- Dependency Injection
  - Techniques
    - Constructor
    - Setter
    - Method

🏭 Creational Patterns

- Dependency Injection

```php
abstract class Bait {}
class Worm extends Bait {}
class Fly extends Bait {}
class SpinnerLure extends Bait {}

class FishingPole
{
    public function __construct(public Bait $bait) {}

    public function setBait(Bait $bait): void
    {
        $this->bait = $bait;
    }

    public function castBait(Bait $bait): void
    {
        $this->bait = $bait;
    }
}

$pole = new FishingPole(new Worm());
$pole->setBait(new Fly());
$pole->castBait(new SpinnerLure());
```

🏭 Creational Patterns

- Factory Pattern
  - Responsible for all class instantiations
  - “createThing” method naming

🏭 Creational Patterns

- Factory Pattern
  - Techniques
    - Creation Method: method in class that creates a Thing
    - Static Creation Method: static create method for a Thing

🏭 Creational Patterns

- Factory Pattern
  - Simple Factory Pattern: creates Things

```
class Thing {}
```

```
class ThingFactory
{
    public function createThing(): Thing
    {
        return new Thing();
    }
}
```

🏭 Creational Patterns

- Factory Pattern
  - Factory Method: interface for creation, uses inheritance
  - Abstract Factory: produce related Things using inputs

```
abstract class Bike {}

class RoadBike extends Bike {}

interface BikeFactoryInterface
{
    public function createBike(): Bike;
}

abstract class BikeFactoryAbstract implements BikeFactoryInterface {}

class RoadBikeFactory extends BikeFactoryAbstract
{
    public function createBike(): RoadBike
    {
        return new RoadBike();
    }
}
```

🏭 Creational Patterns

- Singleton Pattern
  - Ensure one instance only
  - Global state
  - “::getInstance” method
  - Can be used with Lazy Initialization Pattern
    - Delayed object creation

```php
class TakeOutGarbage extends TaskAbstract
{
    public function __construct(protected ?Teenager $teenager = null) {}

    protected function getTeenager(): Teenager
    {
        $this->teenager ??= Teenager::getInstance();
        return $this->teenager;
    }

    public function doTask(): void
    {
        $this->getTeenager()->completeTask($this);
    }
}
```

🏭 Creational Patterns

- Singleton Pattern
  - Use cautiously, can be easily abused
    - Do not have state on a singleton class
    - Violates “S” in SOLID
    - Cannot (easily) test static functions

🏭 Creational Patterns

```php
abstract class SingletonAbstract
{
    protected static SingletonAbstract $instance;

    protected function __construct() {}

    public static function getInstance(): static
    {
        static::$instance ??= new static();
        return static::$instance;
    }

    final protected function __clone() {}

    final protected function __wakeup() {}
}

class SomeService extends SingletonAbstract {}
$someService = SomeService::getInstance();
```

🏭 Creational Patterns

- Singleton
  - Good or bad?

🏢 Structural Patterns

🏢 Structural Patterns

- Make design decisions easier
- Relationships

🏢 Structural Patterns

- Adapter
- Facade

🏢 Structural Patterns

- Adapter
  - Centralized access to library or other classes
    - Rest Client
    - Filesystem
  - Translates (adapts) parameters to compatible format
  - Makes upgrading libraries, framework upgrades easier

```php
class EmailAdapter implements AdapterInterface
{
    public function mail(
        string $to,
        string $subject,
        string $message
    ): bool {
        return mail($to, $subject, $message);
    }
}
```

🏢 Structural Patterns

- Facade
  - Simple interface with complex library or framework
  - Works with more things than an Adapter

```php
class TicketDto
{
    public function __construct(
        public ?int $id,
        public string $title,
    ) {}
}

class TicketingFacade
{
    public function createTicket(TicketDto $ticketDto): TicketDto
    {
        // creates a ticket in Ticketing System
        return $ticketDto;
    }
}
```

⚙️ Behavioral Patterns

⚙️ Behavioral Patterns

- Strategy
- Iterator
- Observer

⚙️ Behavioral Patterns

- Strategy Pattern
  - Interface implemented by different classes
  - Encapsulated logic in the implementing classes
  - Classes using these classes do not need to know how it was implemented

⚙️ Behavioral Patterns

```php
interface Logger
{
    public function log(string $message): void;
    public function error(string $message): void;
}

class SysLogger implements Logger
{
    public function log(string $message): void {}
    public function error(string $message): void {}
}

class DbLogger implements Logger
{
    public function log(string $message): void {}
    public function error(string $message): void {}
}

class QueueWorker
{
    public function __construct(protected Logger $logger) {}

    public function handle(Command $cmd): bool
    {
        $this->logger->log('Starting: ' . $cmd->getName());
        return true;
    }
}
```

⚙️ Behavioral Patterns

- Iterator
  - Used with Collections
  - Hides implementation
  - Use methods for access

⚙️ Behavioral Patterns

```php
abstract class CollectionAbstract implements Iterator
{
    protected array $items = [];
    protected int $key = 0;

    public function current(): EntityInterface
    {
        return $this->items[$this->key];
    }
    public function next(): void
    {
        ++$this->key;
    }
    public function key(): int
    {
        return $this->key;
    }
    public function valid(): bool
    {
        return isset($this->items[$this->key]);
    }
    public function rewind(): void
    {
        $this->key = 0;
    }
    public function add(EntityInterface $entity): void
    {
        $this->next();
        $this->items[$this->key] = $entity;
    }
}
```

⚙️ Behavioral Patterns

- Observer
  - Also termed as Publish and Subscribe
  - Event notifies Observers
  - Observers perform appropriate action
  - Easy to change Observers

⚙️ Behavioral Patterns

```php
interface Observer
{
    public function notify(string $event, string $item): void;
}

class FileArchiver implements Observer
{
    public function notify(string $event, string $item): void
    {
        // archive file
    }
}
```

```php
class FileUpload
{
    /** @var Observer[] */
    protected array $observers;
    public function __construct(array $observers)
    {
        $this->observers = $observers;
    }

    public function upload(string $filename): bool
    {
        // uploads a file
        foreach ($this->observers as $observer) {
            $observer->notify('file:upload', $filename);
        }

        return true;
    }
}
```

💻 Other Patterns

💻 Other Patterns

- Concurrency Patterns
- Service Locator
- Repository

💻 Other Patterns

- Concurrency Patterns
  - Thread safety
  - Transactions
  - Resource lock

💻 Other Patterns

- Service Locator
  - Service Manager
  - Registry of classes
  - Knows how to instantiate classes

```php
class ServiceLocator
{
    public function __construct(protected array $config) {}

    public function get(string $className): object
    {
        return new $className();
    }
}

$serviceLocator = new ServiceLocator([
    UserService::class => [UserRepository::class],
    UserRepository::class => [DbAdapter::class],
]);
$userService = $serviceLocator->get(UserService::class);
```

💻 Other Patterns

- Repository
  - Abstraction layer for data
    - DB, Filesystem, …
  - Only contains data interactions
  - Have supporting Service and Entity
  - Use Abstracts and Interfaces to define behaviors

```php
interface RepositoryInterface
{
    public function insert(EntityInterface $entity): int;
}
class DbAdapter
{
    public function insert(array $record): int
    {
        // inserts a record
        return 1;
    }
}

abstract class RepositoryAbstract implements RepositoryInterface
{
    public function __construct(protected DbAdapter $dbAdapter) {}

    public function insert(EntityInterface $entity): int
    {
        $this->dbAdapter->insert($entity->toArray());
        return 1;
    }
}

class UserEntity implements EntityInterface
{
    public function __construct(
        protected ?int $userId,
        protected string $username
    ) {}

    public function toArray(): array
    {
        return [
            'user_id' => $this->userId,
            'username' => $this->username,
        ];
    }
}

class UserRepository extends RepositoryAbstract {}

class UserService
{
    public function __construct(protected UserRepository $repository) {}

    public function create(UserEntity $userEntity): int
    {
        return $this->repository->insert($userEntity);
    }
}
```

🤓 Benefits

🤓 Benefits

- Proven patterns that work
  - Repeatable
  - Flexible
  - Reduced Refactoring

🤓 Benefits

- Promotes SOLID
  - Single-Responsibility
  - Open-Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion

🤓 Benefits

- Testing
  - Code Testability
  - Code Coverage

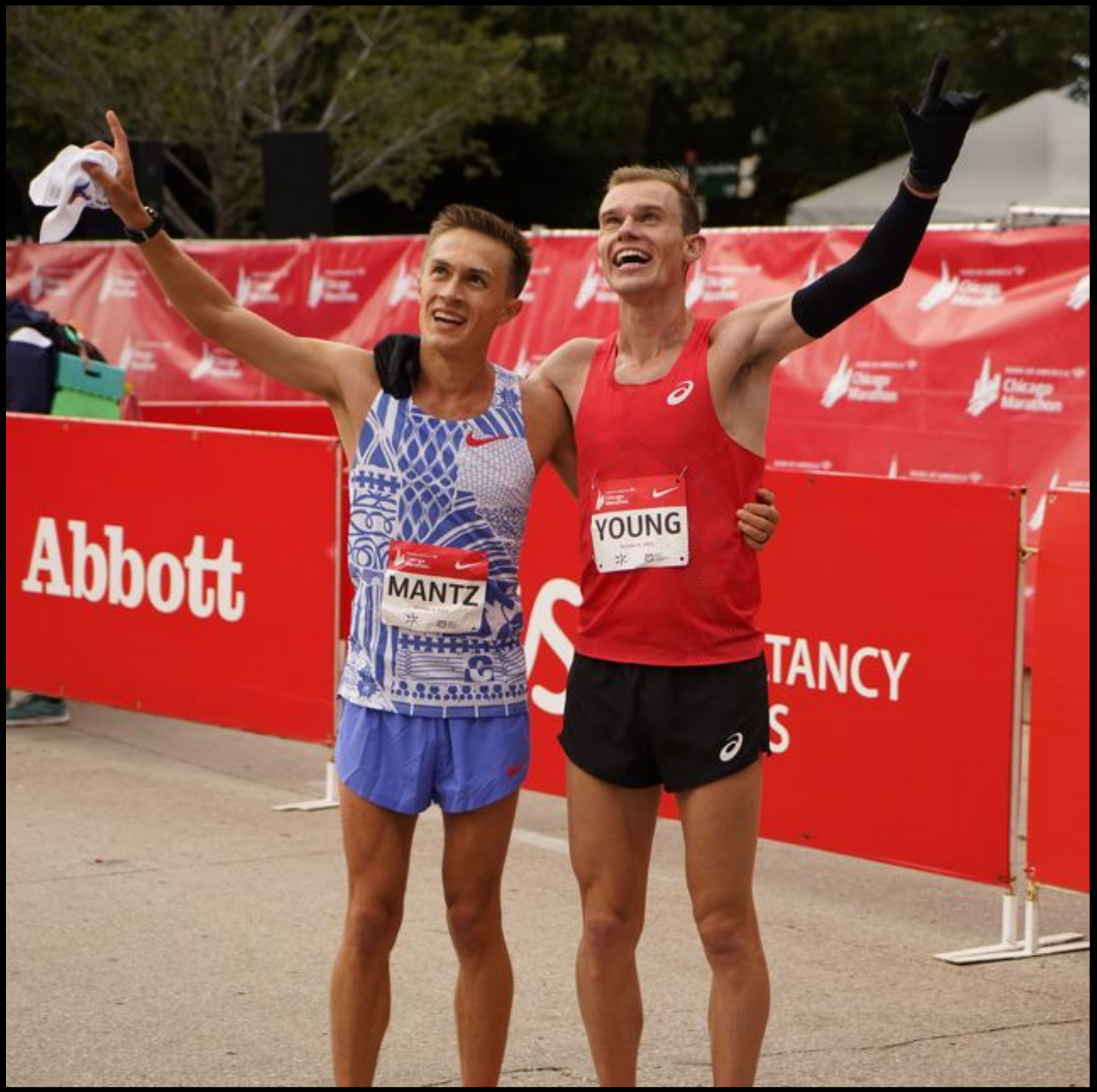


Photo by Derek Call

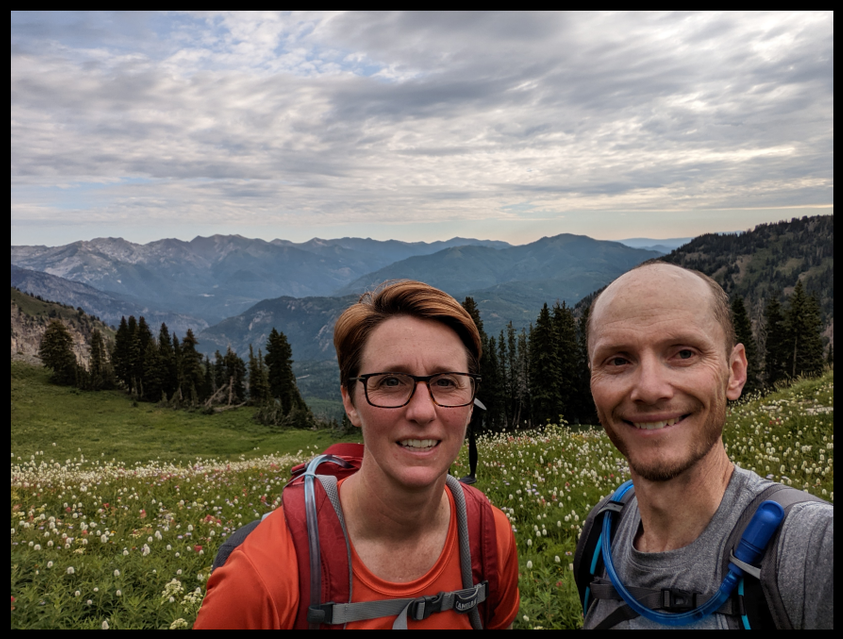
🗣️ Discussion

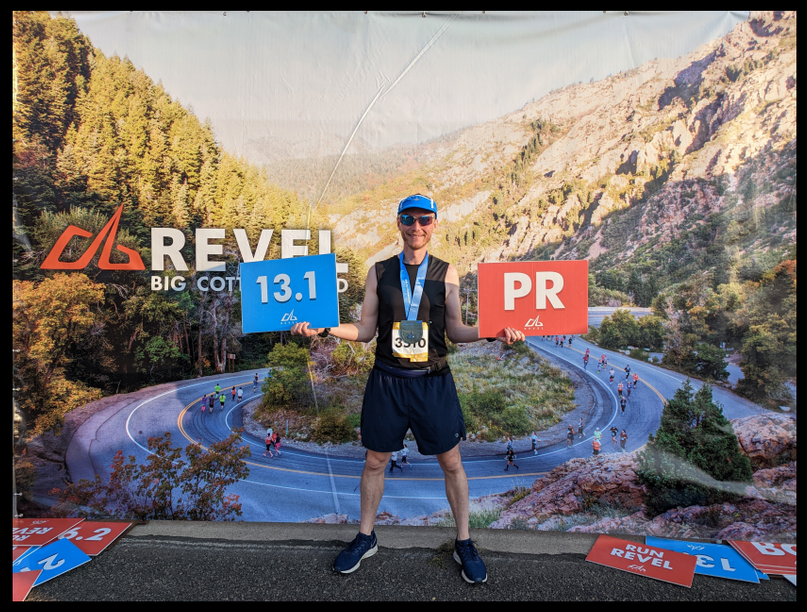
📋 Review

- Pattern Classifications
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
  - Other Patterns
- Benefits

Mark Niebergall @mbniebergall

- PHP since 2005
- Masters degree in MIS
- Senior Software Engineer, Team Lead
- Vulnerability Management project (security scans)
- Utah PHP Co-Organizer, FIG Secretary
- CSSLP, SSCP Certified and Exam Developer
- Endurance sports, outdoors





Architecture Design Patterns for PHP

- Questions?

👀 References

- https://refactoring.guru/design-patterns/php
- https://phptherightway.com/pages/Design-Patterns.html
- https://en.wikipedia.org/wiki/Software_design_pattern