Improving code quality

with

- **Domain-Driven Design**
- **Algebraic Data Types**

Author: Jade Gu

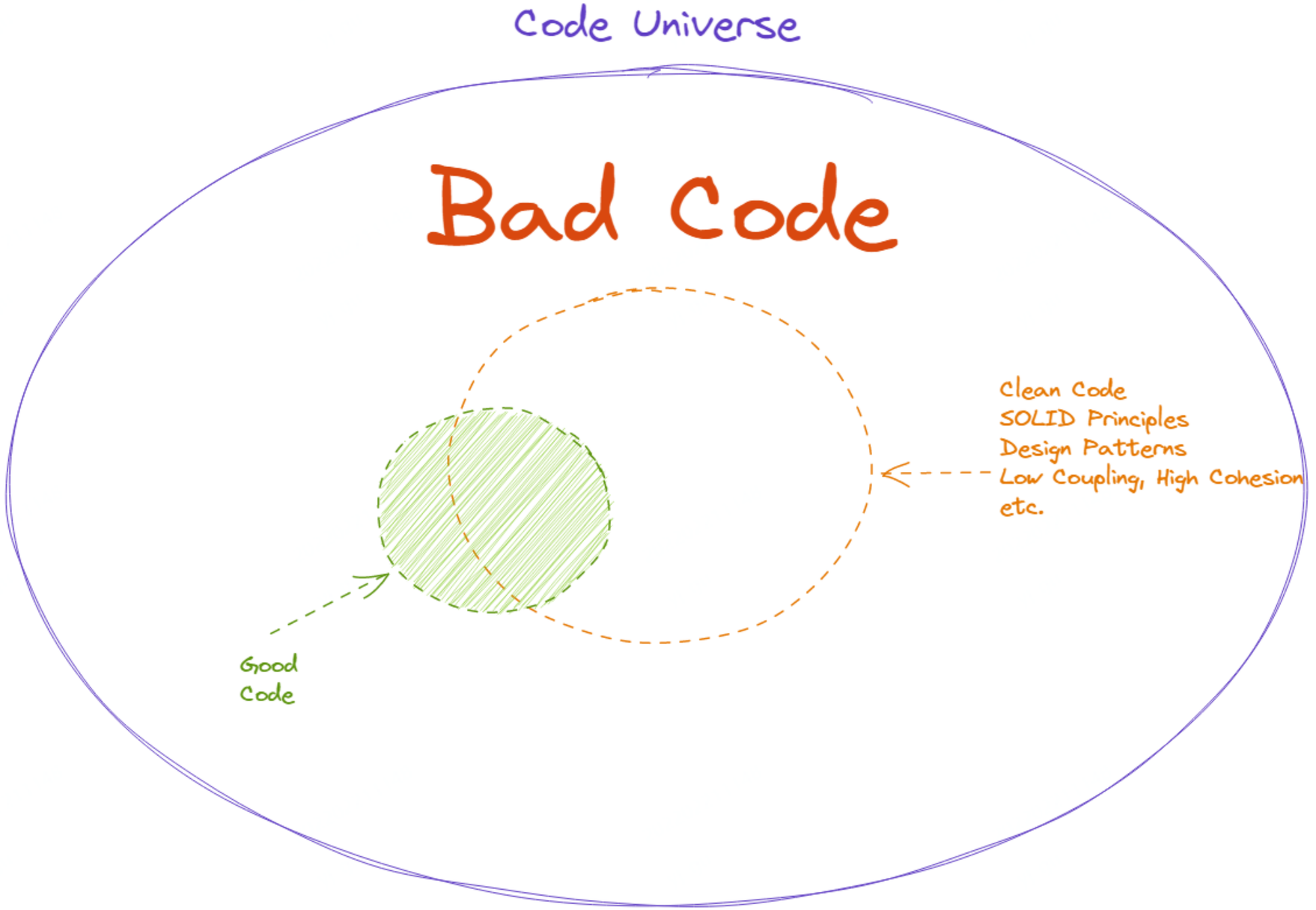
'If you can’t measure it, you can’t improve it.'

代码质量的评估标准

- 可拓展性(Extensibility)
- 可维护性(Maintainability)
- 可读性(Readability)
- 可测试性(Testability)
- etc.

提升代码质量的方法?

- SOLID Principles
- Clean Code Principles
- Design Patterns
- Low Coupling, High Cohesion
- etc.


Code Universe
Bad Code
Clean Code
SOLID Principles
Design Patterns
Low Coupling, High Cohesion
etc.
Good
Code

当前代码指南的不足之处

- Subjective，依赖开发者主观经验
- Unclear，表述模糊，不够清晰
- Hindsight，对已写就的代码做事后评估，对写代码本身缺乏建设性指导
- Imprecise，不够精确，不够准确
- External，围绕代码表面的形式，忽视问题的本质特征，或者假设问题已经被解决
- etc.

我们需要的

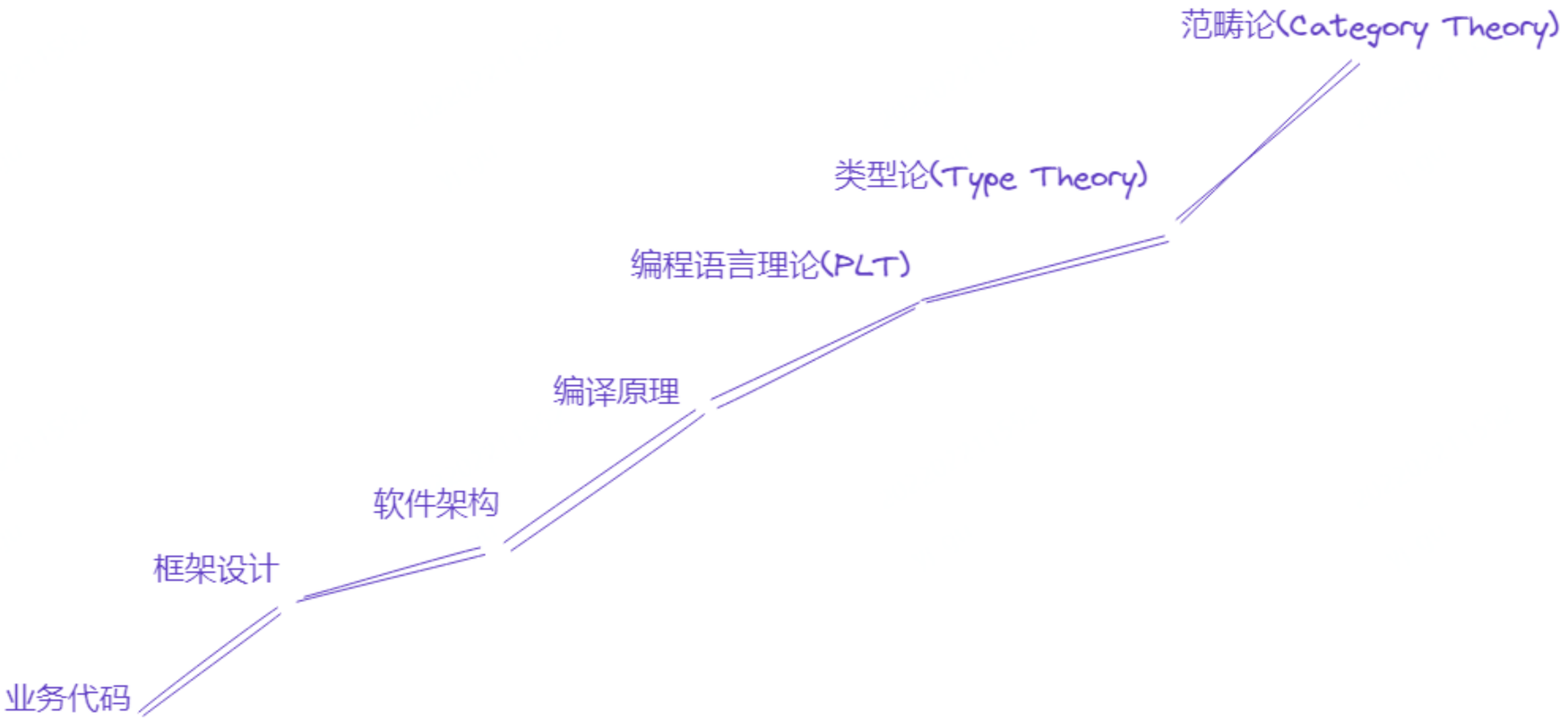
- Objective，更加客观的，所有理性的开发者都有一致的认知
- Clear，表述清晰明确
- Insight，在写代码之前或写代码之时就能帮助洞察问题
- Precise，精确的代码评估标准
- Internal，围绕问题本质出发，不仅仅是代码的编写形式
- etc.

一种代码认知的发展阶段性

- 看山是山：写简单的代码，简单地写代码，简单但平凡
- 看山不是山：熟练搭建各种代码架构，熟练运用各种设计模式
- 看山还是山：写简单的代码，简单地写代码，简单但不平凡

另一种代码认知的发展阶段性

一山还有一山高：学术化、数学化是必由之路

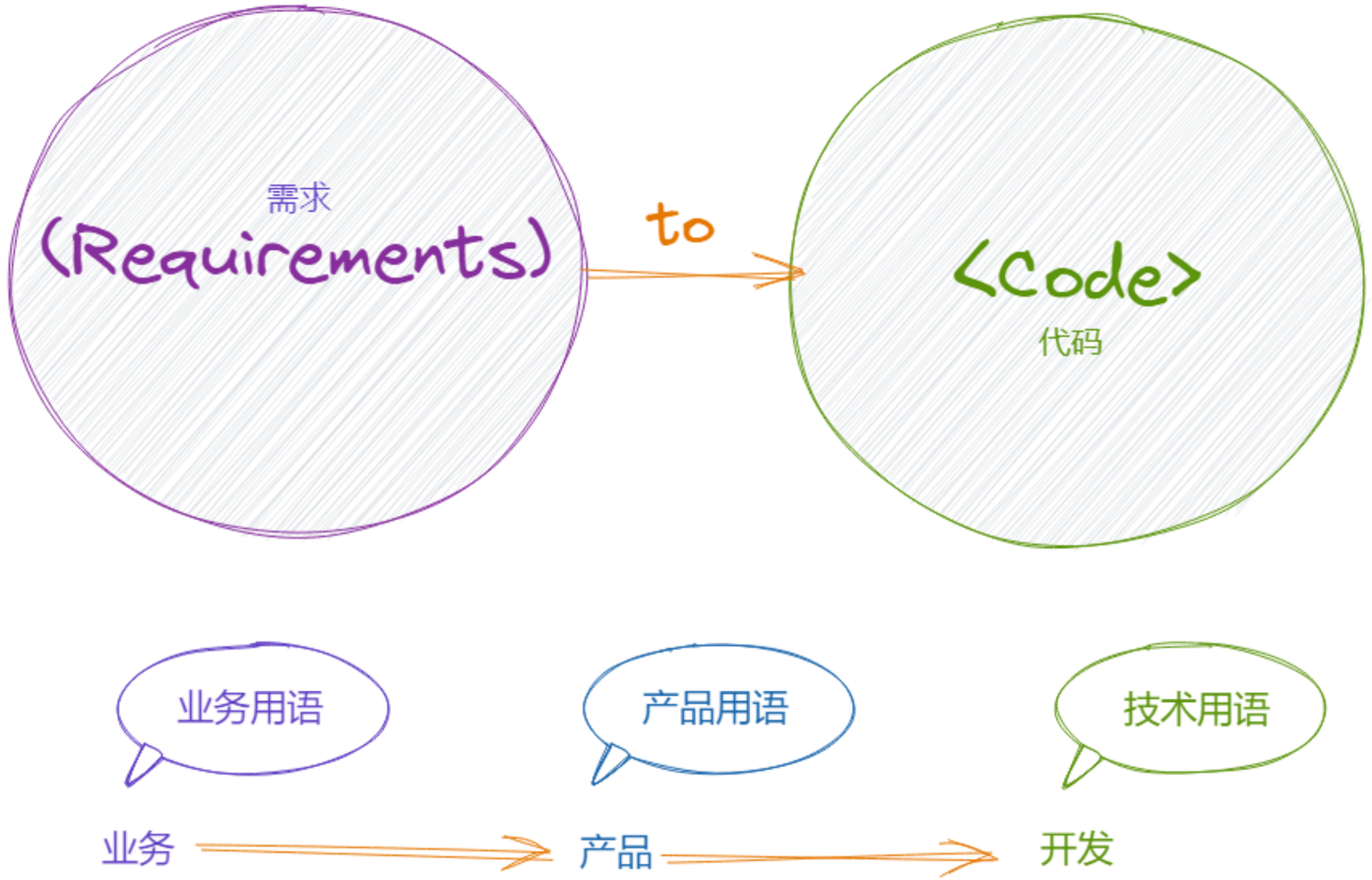


DDD 和 ADT 的联姻

Domain-Driven Design

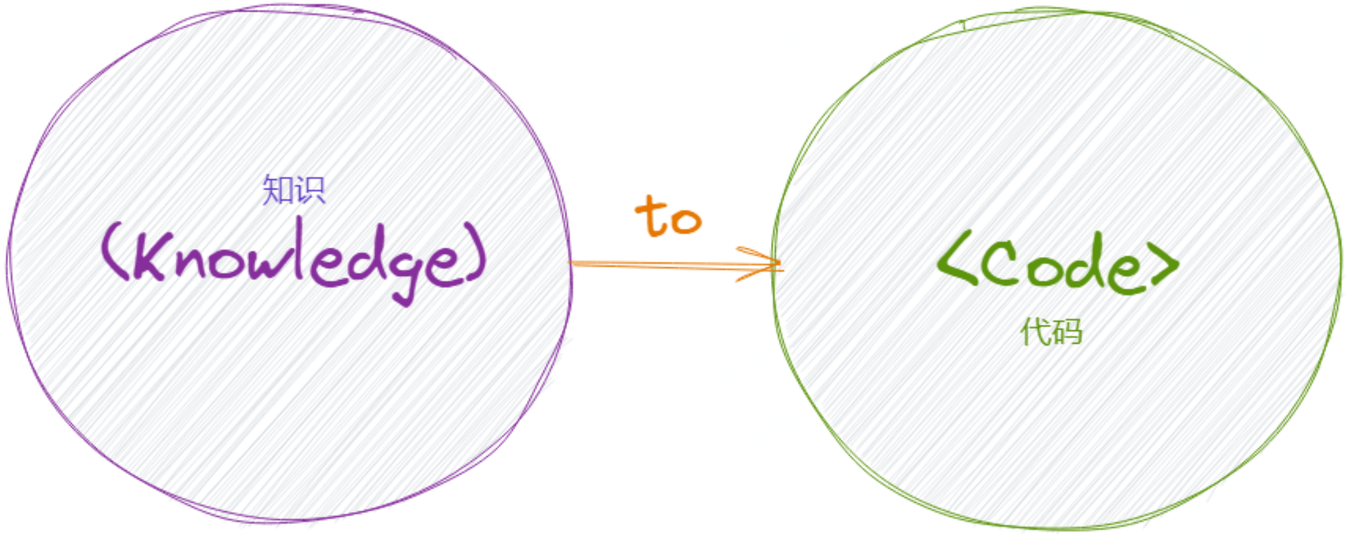
Algebraic Data Types

朴素产研模型：需求驱动开发



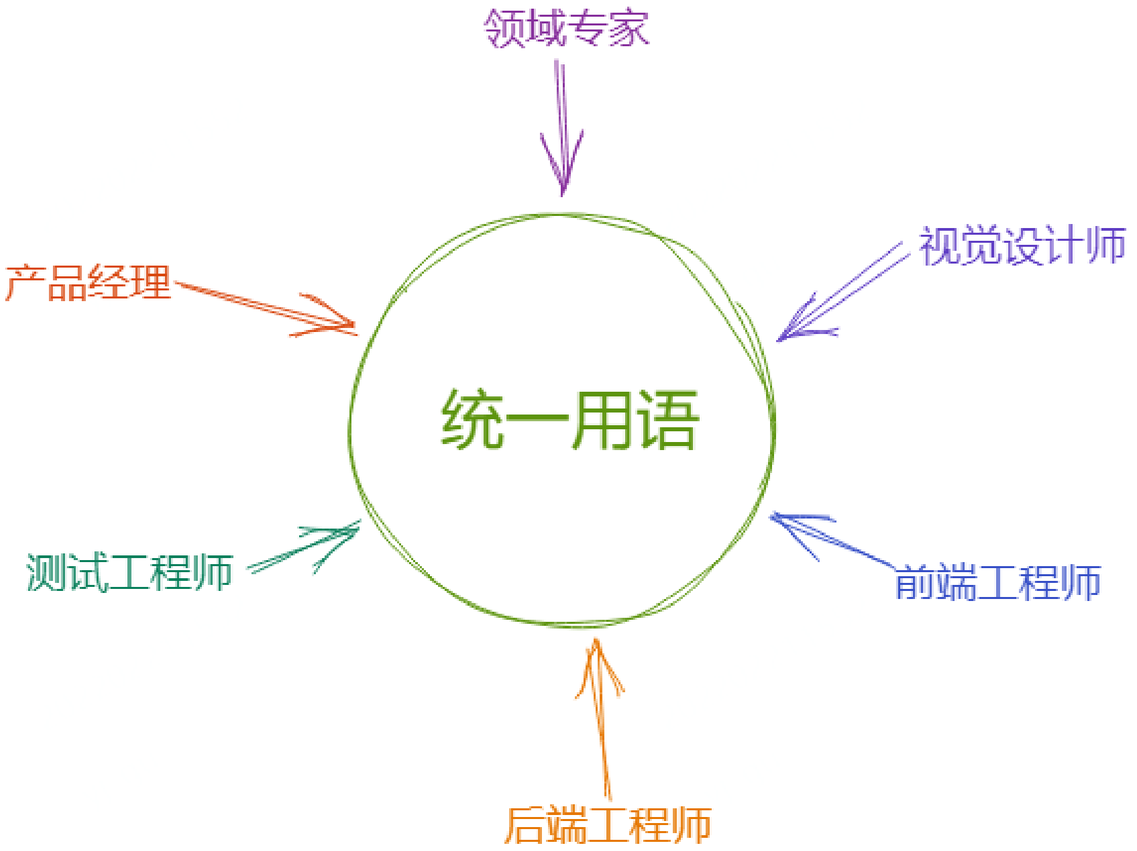
领域驱动设计的核心思想

从领域知识到代码实现的转换



领域驱动设计的关键过程

构建团队统一用语，形成领域共识



需求驱动开发的会议场景

领域驱动设计的会议场景

事件风暴(Event Storming)

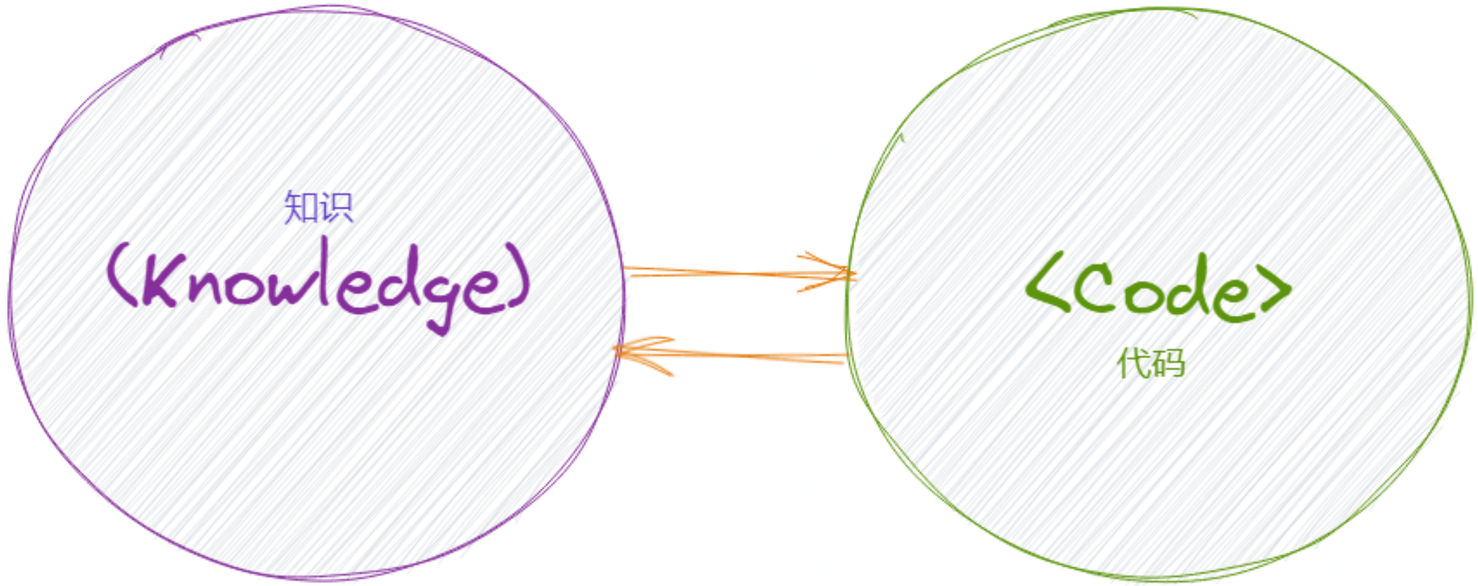
小结

关键过程：领域知识的准确传递

- 高质量的代码来自对问题的正确认知，很难在不理解问题的基础上优雅地解决问题
- 代码的写法、风格、模式等手段，建立在正确的认知基础上才能达到最佳的效果
- 忽视提高对问题的认知水平，盲目地运用代码技巧、设计模式，往往让代码更糟糕
- DDD 的核心思想是，以领域专家为核心，建立统一用语，确保知识和需求的可靠传递

领域驱动设计的战术精髓

代码应当忠诚地反映领域知识



领域和领域知识的定义

- Domain(领域)是一系列关联问题构成的集合
- Domain Knowledge(领域知识)是一系列关联问题涉及的所有**真命题**(true proposition)的集合

Curry–Howard isomorphism

用代码表达领域知识的原理——柯里-霍华德同构

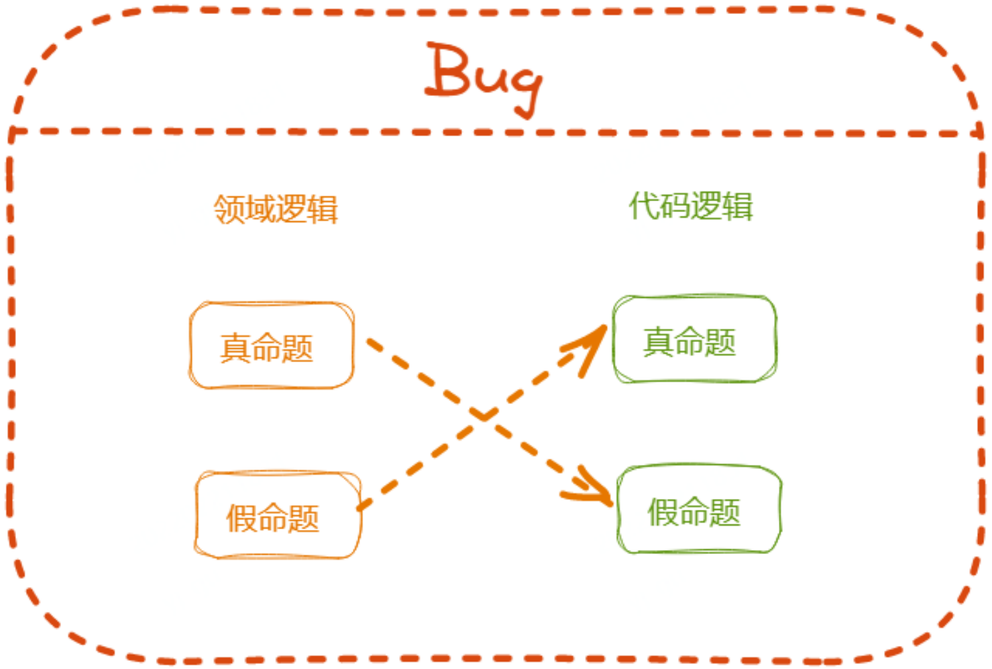
- 命题即类型，证明即程序(Propositions as Types, Proofs as Programs)
- Type-Driven Development(类型驱动开发)，用类型去表达领域知识(领域里的真命题)
- 符合类型的所有值，都是该类型所表征的命题的证明(Witness)
- 真命题：至少有一个值的类型
- 假命题：没有任何值的类型

Logic Name	Logic Notation	Type Notation	Type Name
True	$\top$	$\top$	Unit Type
False	$\bot$	$\bot$	Empty Type
Not	$\neg A$	$A \to \bot$	Function to Empty Type
Implication	$A \to B$	$A \to B$	Function
And	$A \wedge B$	$A \times B$	Product Type
Or	$A \vee B$	$A + B$	Sum Type
For All	$\forall a \in A, P(a)$	Π a : A . P(a)	Dependent Function
Exists	$\exists a \in A, P(a)$	Σ a : A . P(a)	Dependent Product Type

Bug 的定义

领域里的命题跟代码里的命题不一致 === Bug

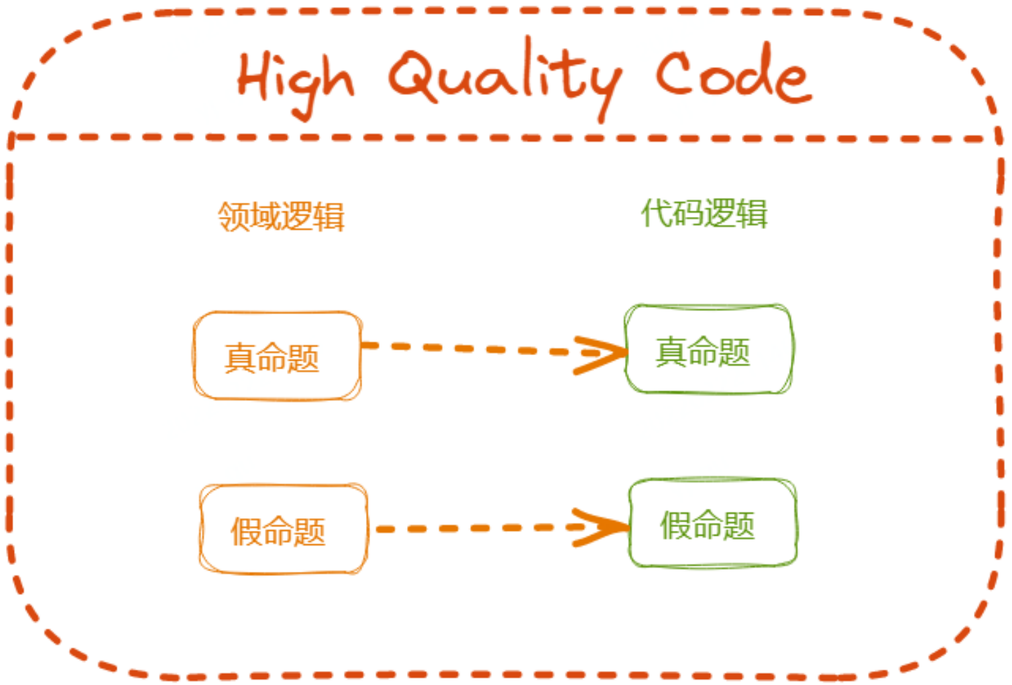
- 领域里的真命题(领域知识)，代码里却是假命题(Error/Crash/Halt)
- 领域里的假命题，代码里却是真命题(Illegal-State/Unexpected-Behavior)



高质量的代码的判别标准

领域里的命题跟代码里的命题一致 === High Quality Code

- 领域里的真命题(领域知识)，代码里也是真命题
- 领域里的假命题，代码里也是假命题



Type Theory 基础知识

In type theory, every term has a type. A term and its type are often written together as "term : type".

- **Empty type**: The empty type has no terms.——0
- **Unit type**: The unit type has one term.——1
- **Boolean type**: The boolean type has two terms, true and false.——2
- **Natural Numbers type**: The natural numbers type has infinite terms which are the integers with non-negative values.——Infinite
- etc.

Algebraic Data Types

In type theory, an algebraic data type is a kind of composite type, i.e., a type formed by combining other types.

- "sum" is alternation (A | B, meaning A or B but not both)
- Sum type: `size(A | B) = size(A) + size(B)`
- "product" is combination (A & B, meaning A and B together),
- Product type: `size(A & B) = size(A) * size(B)`

```ts
// sum types
type Value = string | number;

const a: Value = 'John';
const b: Value = 70;
```

```ts
// product types
type Person = { name: string; age: number };
// type Person = { name: string } & { age: number }

const person: Person = { name: 'John', age: 70 };
```

实战案例一：用户信息

领域规则(Domain rules)

- 用户要么是已登录用户，要么是未登录用户（游客）
- 游客拥有随机的昵称
- 已登录用户拥有昵称、Email 信息
- Email 信息要么是已验证的 Email，要么是未验证的 Email
- 已验证的 Email 有验证时间戳
- 用户信息通过 Http API 获取

实战案例一：用户信息之 Domain types

常见的领域类型建模-大道至简

```ts
type UserInfo = {
  // 当用户未登录时，id 为空字符串
  id: string;
  // 当用户未登录时，name 为随机生成的昵称
  name: string;
  // 当用户未登录时，email 为空字符串
  email: string;
  // 用户是否登录
  isLogin: boolean;
  // 当邮箱未验证时，这个字段为 false
  isEmailVerified: boolean;
  // 当邮箱已验证时，这个字段为验证时间戳，否则为空字符串
  emailVerifiedAt: string;
};

// Http API 获取用户信息
type JsonResponse = {
  error?: string;
  // 当 error 为空时，这个字段为用户信息
  data?: UserInfo;
};
```

实战案例一：用户信息之 Domain types

基于 DDD + ADT 的领域类型建模

EmailInfo types

```
type VerifiedEmailInfo = {
  type: 'VerifiedEmailInfo';
  email: string;
  verifiedAt: string;
};

type UnverifiedEmailInfo = {
  type: 'UnverifiedEmailInfo';
  email: string;
};

type EmailInfo = VerifiedEmailInfo | UnverifiedEmailInfo;
```

实战案例一：用户信息之 Domain types

基于 DDD + ADT 的领域类型建模

UserInfo types

```
type LoginUserInfo = {
  type: 'LoginUserInfo';
  id: string;
  name: string;
  emailInfo: EmailInfo;
};

type GuestUserInfo = {
  type: 'GuestUserInfo';
  name: string;
};

type UserInfo = LoginUserInfo | GuestUserInfo;
```

实战案例一：用户信息之 Domain types

基于 DDD + ADT 的领域类型建模

JsonResponse types

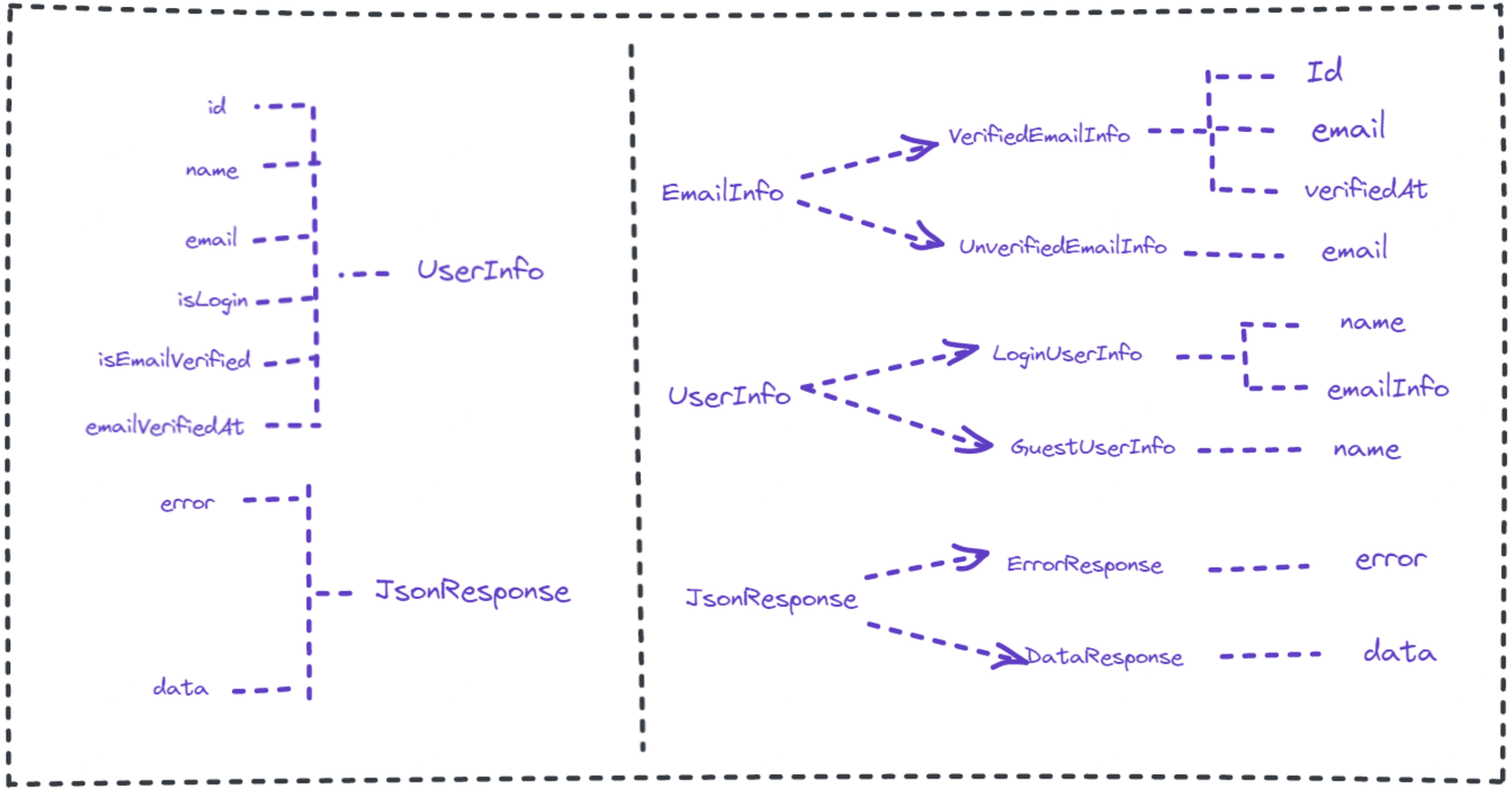
```
type ErrorResponse = {
  type: 'ErrorResponse';
  error: string;
};

type DataResponse = {
  type: 'DataResponse';
  data: UserInfo;
};

type JsonResponse = ErrorResponse | DataResponse;
```

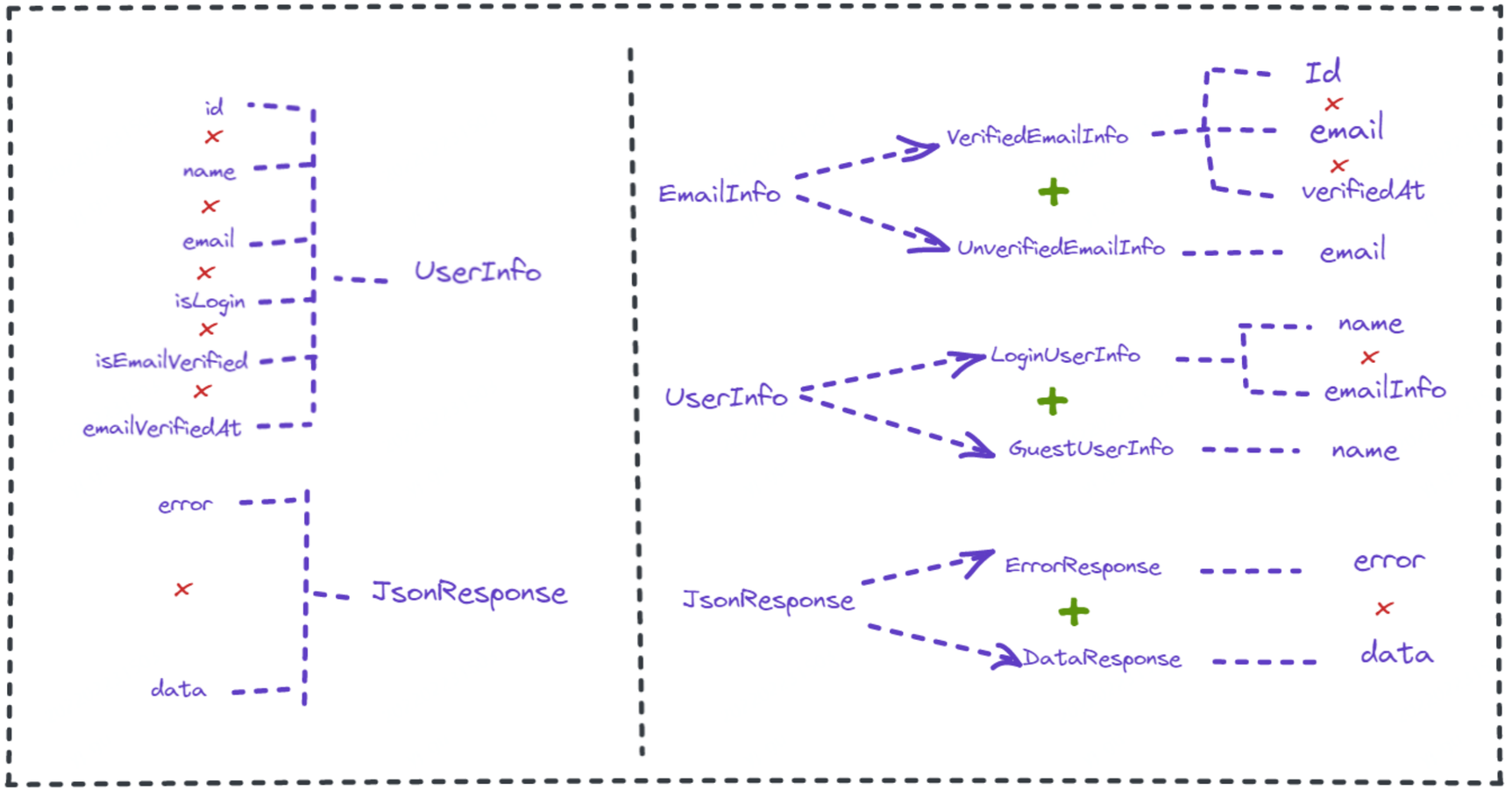
实战案例一：用户信息

表面上的：简单 vs 复杂



实战案例一：用户信息

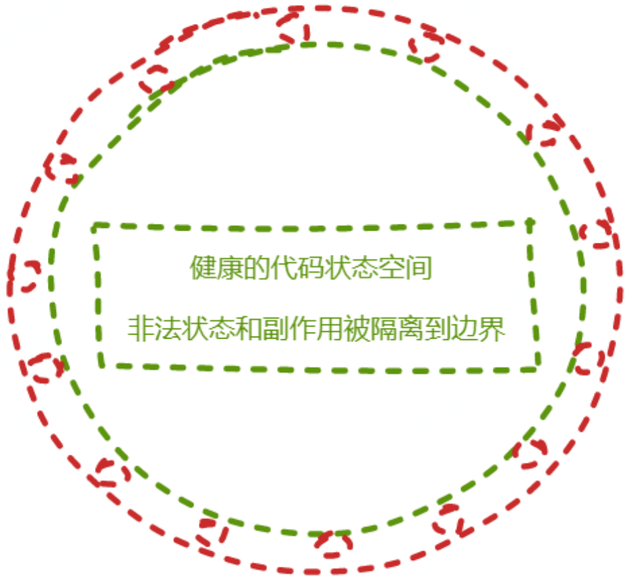
实质的：简单 vs 复杂



实战案例一：用户信息-小结

用 product type 去替换 sum type 的代价

- 领域知识里 `Or` 的关系，被曲解为 `And`，类型由加法复杂度，变成乘法复杂度
- 代码上能写出来的值(value)的数量(terms size)，大于领域知识里的真命题的需求
- 代码里的真命题(多出来的值)，是领域里的假命题，它们成了非法状态（Illegal-States）
- 所有消费数据的地方，都需要做防御性判断，排除非法状态，否则就导致程序出现 BUG
- 系统的可维护性，跟非法状态在代码库里的泄漏程度成反比，泄漏越多，越难以维护和预测



'Making illegal states unrepresentable'

实战案例二：流程控制

领域规则(Domain rules)

- 用户发帖有 3 个阶段：草稿、审核、发布
- 草稿不能跳过审核直接发布
- 草稿可以提交审核
- 审核通过后可以发布
- 审核中的帖子不能修改
- 审核不通过退回草稿阶段

实战案例二：流程控制

常见的领域行为的类型建模-大道至简

```
class Post {
  constructor(private isDraft: boolean, private isReviewing: boolean, private isPublished: boolean, private content: str
  edit(content: string) {
    if (!this.isDraft) {
      throw new Error('Post is not in draft stage');
    }
    this.content = content;
  }
  review() {
    if (!this.isDraft) {
      throw new Error('Post is not in draft stage');
    }
    this.isDraft = false;
    this.isReviewing = true;
  }
  publish() {
    if (!this.isReviewing) {
      throw new Error('Post is not in reviewing stage');
    }
    this.isReviewing = false;
    this.isPublished = true;
  }
  reject() {
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

DraftPost type

```
class DraftPost {
  constructor(private content: string) {}
  edit(content: string) {
    this.content = content;
  }
  review() {
    return new ReviewingPost(this.content);
  }
}
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

ReviewingPost type

```
class ReviewingPost {
  constructor(private content: string) {}
  publish() {
    return new PublishedPost(this.content);
  }
  reject() {
    return new DraftPost(this.content);
  }
  approve() {
    return new PublishedPost(this.content);
  }
}
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

PublishedPost type

```
class PublishedPost {
  constructor(private content: string) {}
  getContent() {
    return this.content;
  }
}
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

```
type DraftPost = {
  type: 'DraftPost';
  content: string;
}

type ReviewingPost = {
  type: 'ReviewingPost';
  content: string;
}

type PublishedPost = {
  type: 'PublishedPost';
  content: string;
}
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

```
const edit = (post: DraftPost, newContent: string): DraftPost => {
  return {
    ...post,
    content: newContent
  }
}

const review = (post: DraftPost): ReviewingPost => {
  return {
    type: 'ReviewingPost',
    content: post.content
  }
}
```

实战案例二：流程控制

基于 DDD + ADT 的领域行为类型建模

```
const approve = (post: ReviewingPost): PublishedPost => {
  return {
    type: 'PublishedPost',
    content: post.content
  }
}

const reject = (post: ReviewingPost): DraftPost => {
  return {
    type: 'DraftPost',
    content: post.content
  }
}
```

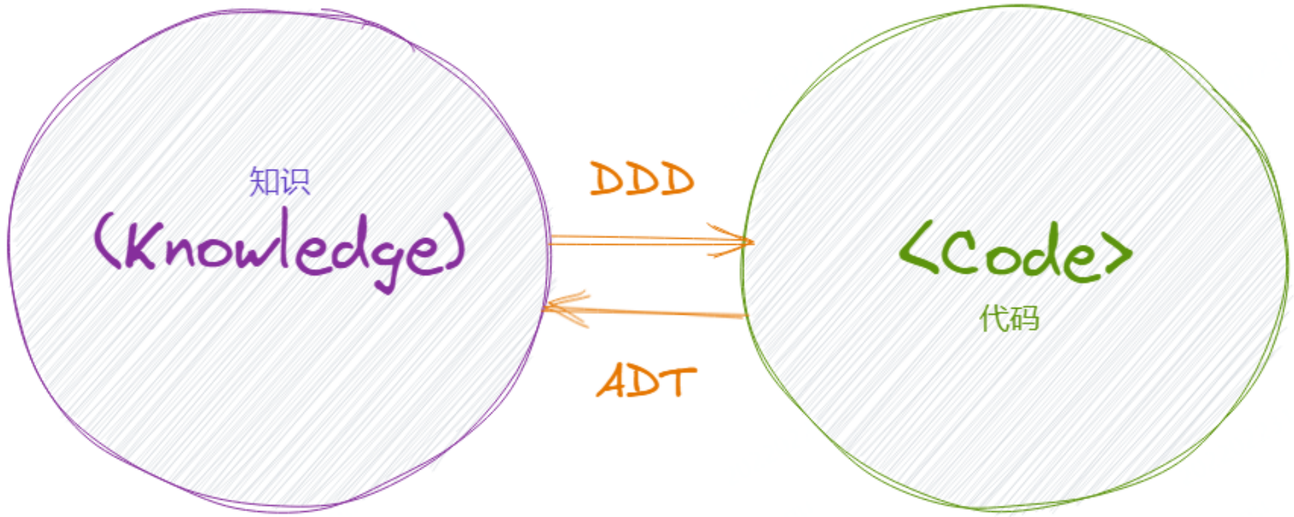
实战案例二：流程控制-小结

- 将**互斥**的操作放到一起**并存**，关系从 `Or` 变成了 `And`，从加法复杂度变成乘法复杂度
- 代码上能调用的函数/方法的数量(terms size)，大于领域知识里的真命题的实际需求
- 代码里的真命题（多出来的方法调用），是领域里的假命题，它们成了非法操作(Illegal-Operations)
- 所有调用方法的地方，都需要做防御性判断，排除非法调用，否则可能导致程序抛错和出 Bug
- 系统的可维护性，跟非法操作在代码库里的泄漏程度成反比，泄漏越多，越难以维护和预测

'Making illegal operations unrepresentable'

总结

- 运用领域驱动设计(DDD)，建立团队统一用语，获得可靠的领域知识，挖掘真实需求
- 运用代数数据类型(ADT)，对领域知识进行一比一建模，获得可靠的代码设计
- DDD+ADT：从知识中可以推导出代码，从代码中可以推导出知识，知识与代码的同构
- 核心技巧：多用 Sum type，少用 Product type，减少非法状态和非法操作的泄漏



目标回顾

一个世纪的学术积累与沉淀，让我们更好的理解领域驱动设计(DDD)，以及代数数据类型(ADT)

- Objective，更加客观的，所有理性的开发者都有一致的认知
- Clear，表述清晰明确
- Insight，在写代码之前或写代码之时就能帮助洞察问题
- Precise，精确的代码评估标准
- Internal，围绕问题本质出发，不仅仅是代码的编写形式