

91APP



# 專案重做的 限創理論與刻意發現 \*

李智樺 / Ruddy Lee (91APP敏捷教練)

# 講師簡介



具有超過 36 年的工程師經歷，擅長帶領研發團隊開發與創新，著作有：《精實開發與看板方法》、《Windows Azure 雲端開發》、《WF工作流程引擎程式設計》、《微軟 VSTS 開發實戰》等書。曾擔任多家資訊公司的研發部經理，為專業的軟體工程顧問、Scrum 及看板課程教學的教練及講師。

# 前言

讓你重做一次剛剛完成的專案

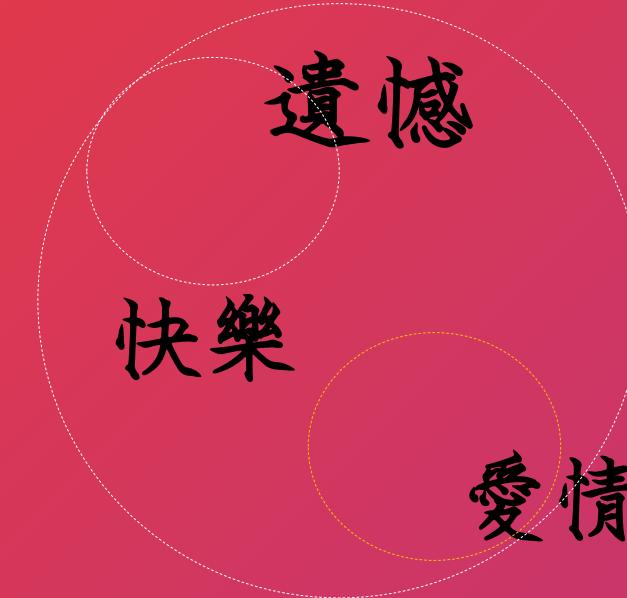
這<sup>此</sup>次<sup>が</sup>我<sup>々</sup>們<sup>ら</sup>會<sup>る</sup>更<sup>々</sup>好<sup>く</sup>地<sup>で</sup>瞭<sup>る</sup>解<sup>せ</sup>...

這<sup>此</sup>次<sup>が</sup>會<sup>る</sup>有<sup>る</sup>所<sup>々</sup>不<sup>同</sup>な<sup>く</sup>

這<sup>此</sup>次<sup>が</sup>我<sup>々</sup>們<sup>ら</sup>會<sup>る</sup>準<sup>時</sup>に<sup>て</sup>完<sup>成</sup>す<sup>可</sup>的<sup>な</sup>

發想

人生；讓你重來一次 ...



發想

讓你重來一次 ...

## 《慶餘年》



范閒 講述一個一出生，就記得  
前世種種的少年的傳奇故事。

專案重來一次

發點時間；想一想

讓你重做一次剛剛完成的專案



沉思者

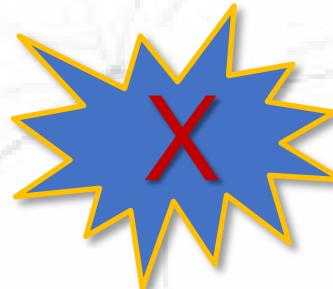
(法)奧古斯特·羅丹

# 想像一下；你必須重做你剛剛完成的項目...

... 完全從頭開始

- 有相同的目標
- 與相同的一群人
- 有相同的約束條件...

重做一次



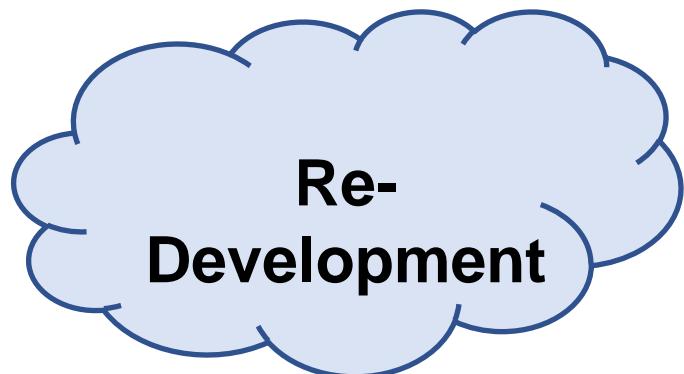
重構

# 想像一下；你必須重做你剛剛完成的項目...

... 完全從頭開始

- 有相同的目標
- 與相同的一群人
- 有相同的約束條件...

~~重構~~



或





無知將是你最大的限制!

Ignorance is the constraint.

- Dan North; BDD 之父

無知：對目的物缺乏相關資訊或是錯誤的認知。

## 二、限制理論

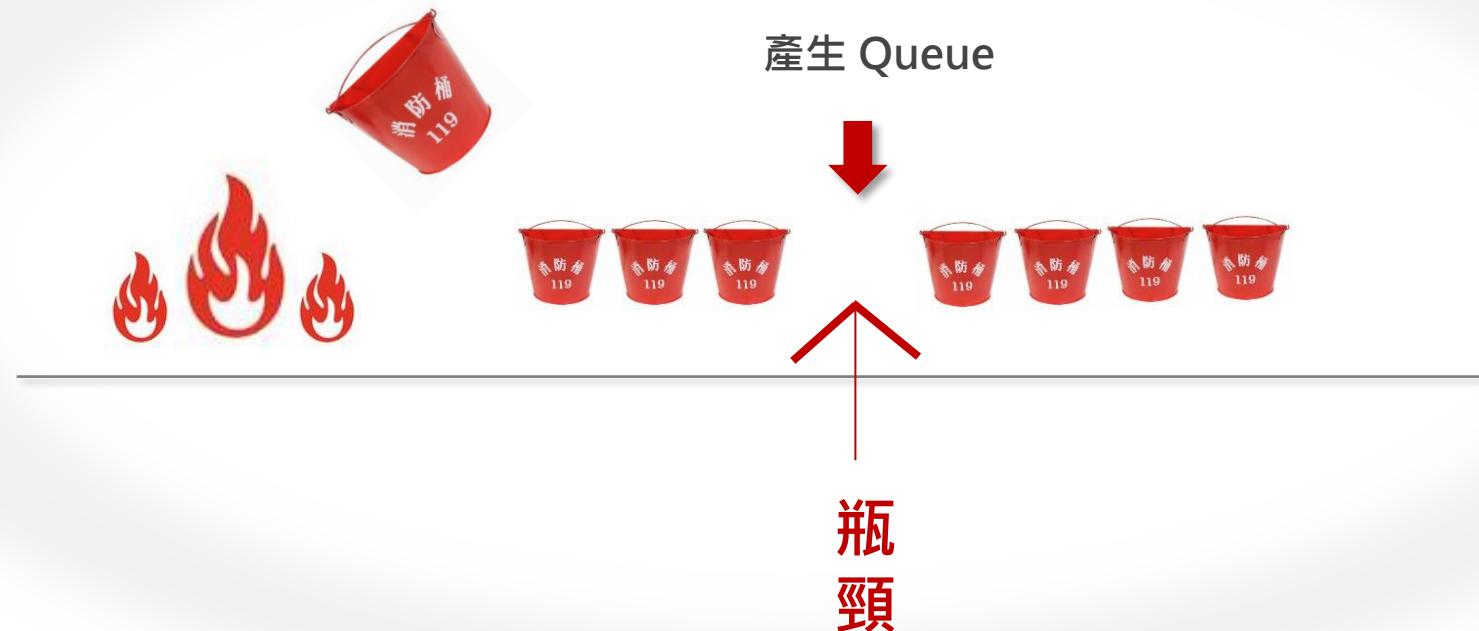
Theory of Constraints



1947 – 2011 艾利·高德拉特 *Eli Goldratt*

# 提水救火!!!

村子裡失火的警鐘響了起來 ...



# 限製理論 Theory of Constraints

高德拉特 所發展出來的一種全方面的管理哲學(思維) , 主張任何系統一定存在著約束(限制)。

系統目前一定存在約束 ...

- 約束背後的任何庫存都是浪費 ,
- 約束背後的任何優化 , 都是沒用的局部優化 ,
- 任何時候不解決約束都是浪費 ,
- ...

→ - 但是你不知道約束在哪裡 !



持續解決瓶頸 ?

# 限制理論 Theory of Constraints

系統一定存在約束.

系統目前一定存在約束 ...

- 約束背後的任何庫存都是浪費 ,
- 約束背後的任何優化 , 都是沒用的局部優化 ,
- 任何時候不解決約束都是浪費 ,
- 但是你不知道約束在哪裡 !



《關鍵鏈》  
Critical Chain

「提水救火」



- 突破專案管理的瓶頸

# 如何解決瓶頸



## 限制理論

步驟一、找出系統瓶頸

橋是瓶頸

步驟二、挖掘瓶頸

挖掘橋的產能

步驟三、遷就瓶頸

遷就於橋，大羊先行、  
Queue 的批次...。

步驟四、打破瓶頸

打破橋的瓶頸 – 擴寬

步驟五、回到步驟1

回到步驟1, 持續改善

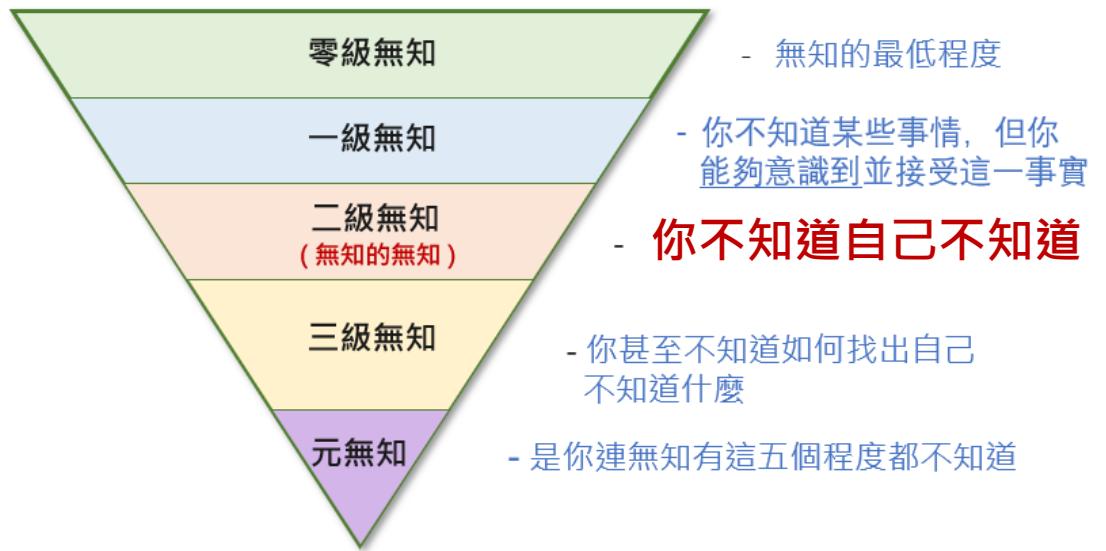
問題: 最慢的一位要放在哪裡?

## 三、無知與模糊不清

Ignorance and ambiguity

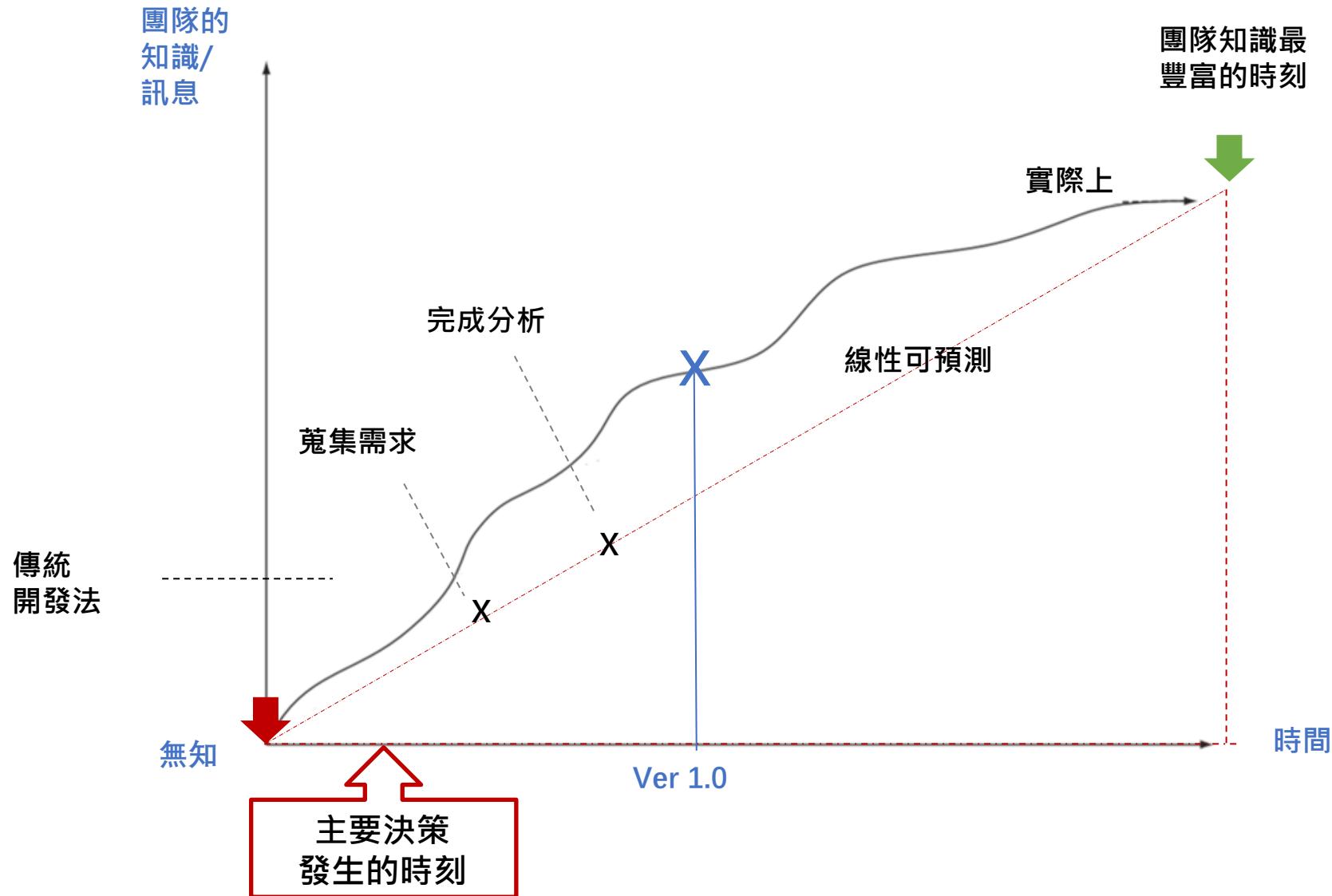
- 敏捷的迷失

### 『無知』的分級

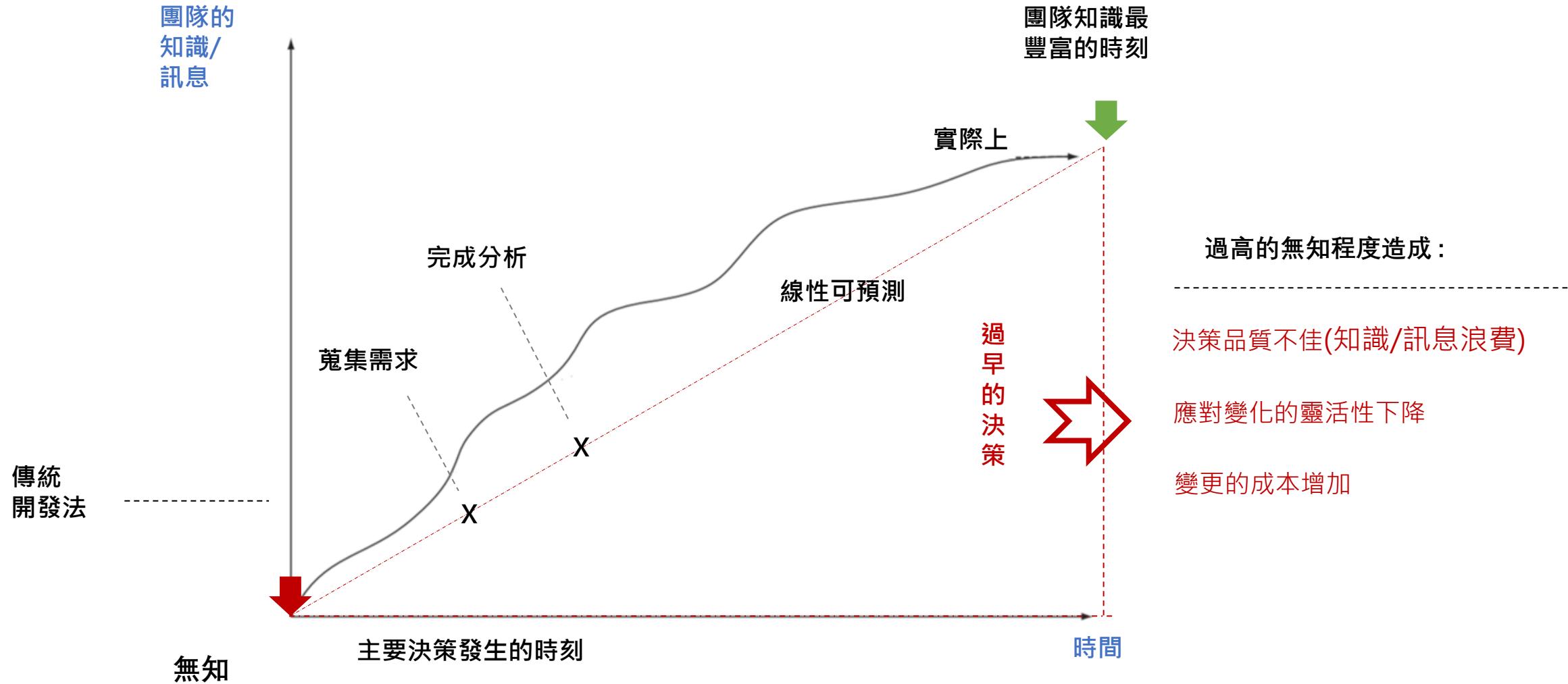


# 產品開發中的本質!

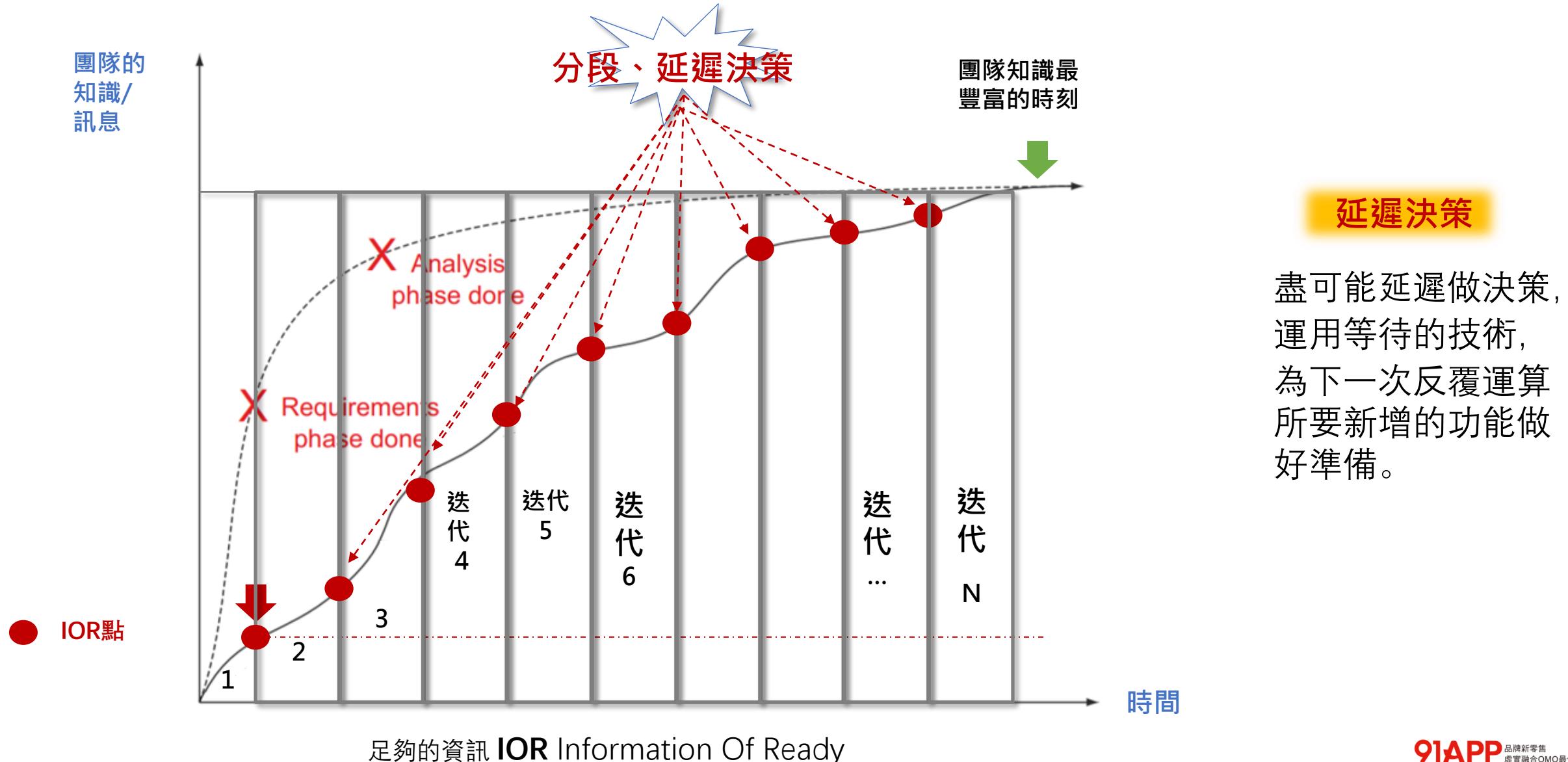
軟體開發是知識獲取的活動。



# 產品開發中的本質!



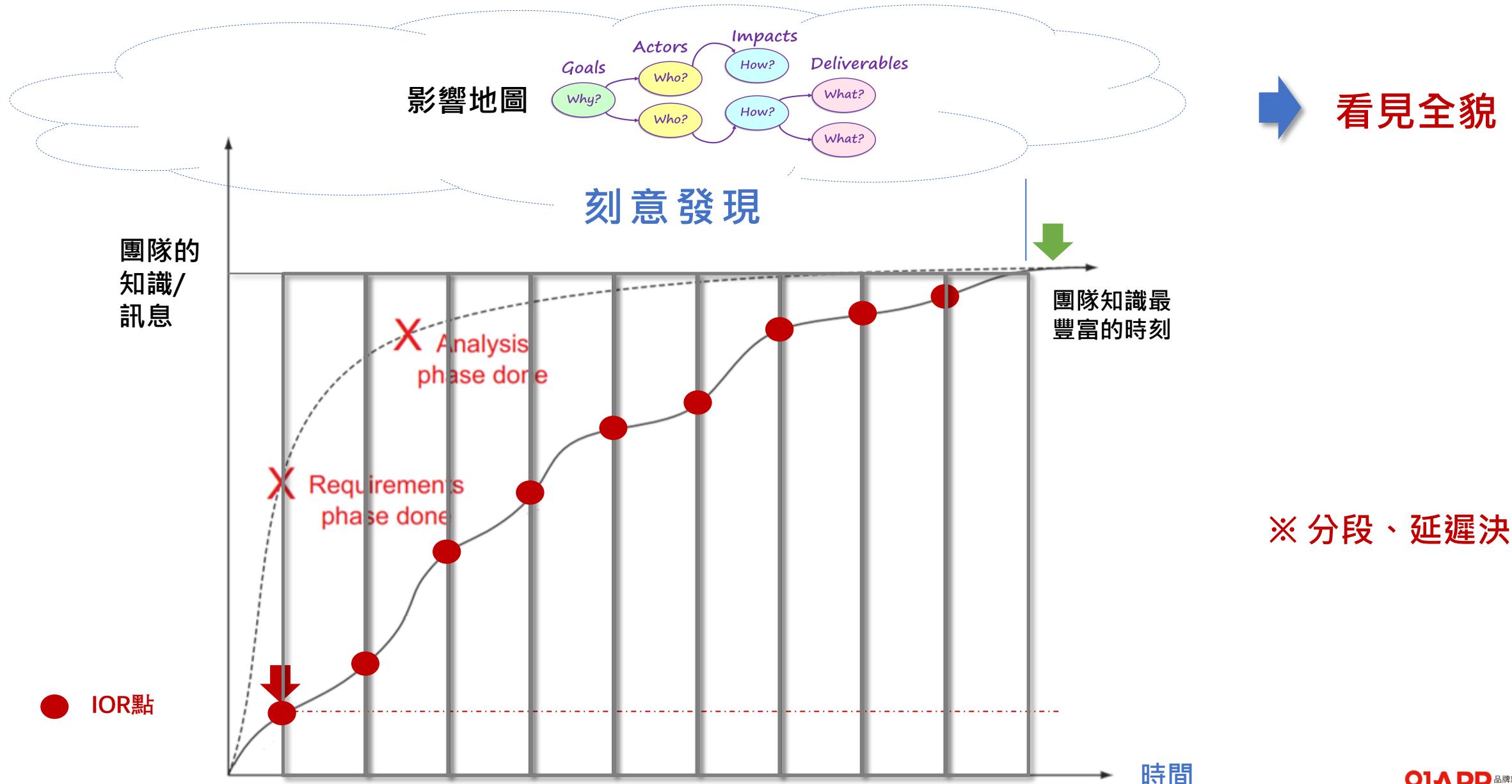
# 敏捷運用小增量、多反覆運算來克服限制!



# 運用工具來進行「刻意發現」！



# 刻意發現 運用影響地圖!



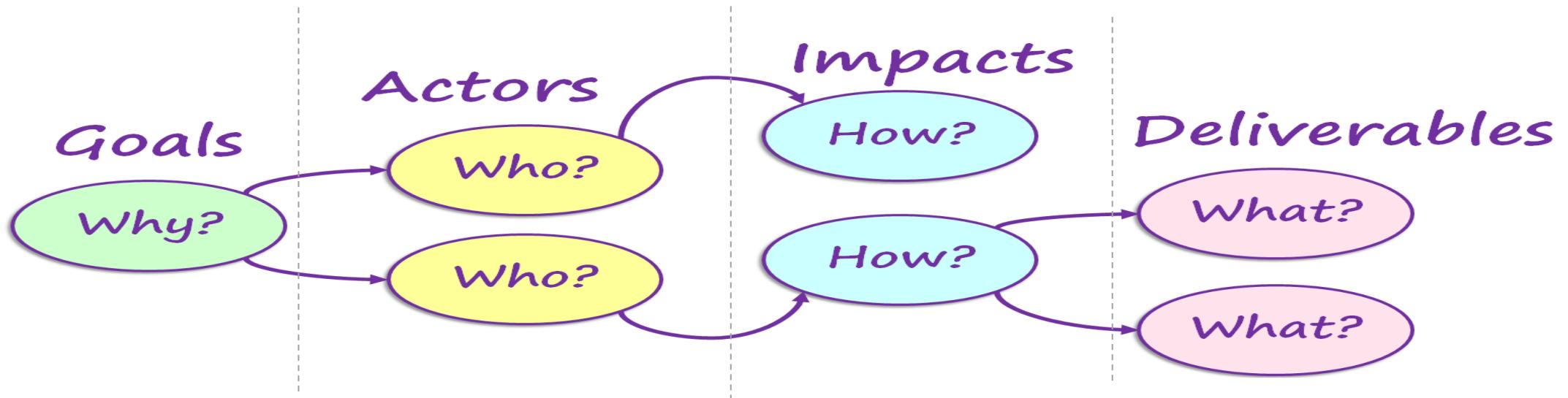
# 解決策略規劃的工具 - 影響地圖

【目標】

【角色】

【影響】

【交付】



為什麼 why ?

誰 Who ?

怎樣 How ?

什麼 What ?

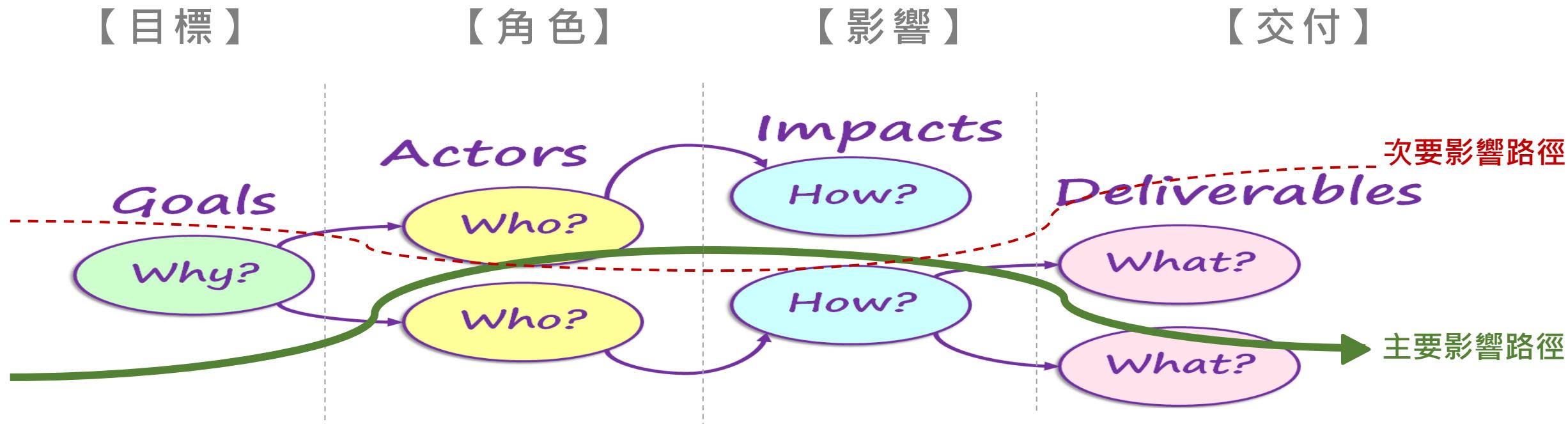
中央部分回答了最重要的問題：  
◆ 我們為什麼做這些？

第一層回答的問題：  
◆ 誰能產生需要的效果？  
◆ 誰會阻礙它？

第二層從業務目標的角度設置角色，它回答如下問題：  
◆ 角色的行為是怎樣改變的？  
◆ 他們怎樣說明我們達成目標？

◆ 影響的不是產品功能，所以要關注業務活動而非關於軟體的想法。

# 解決策略規劃的工具 - 影響地圖



預設替代路徑



里程碑



替代



OKR

目標與關鍵成果

為什麼？

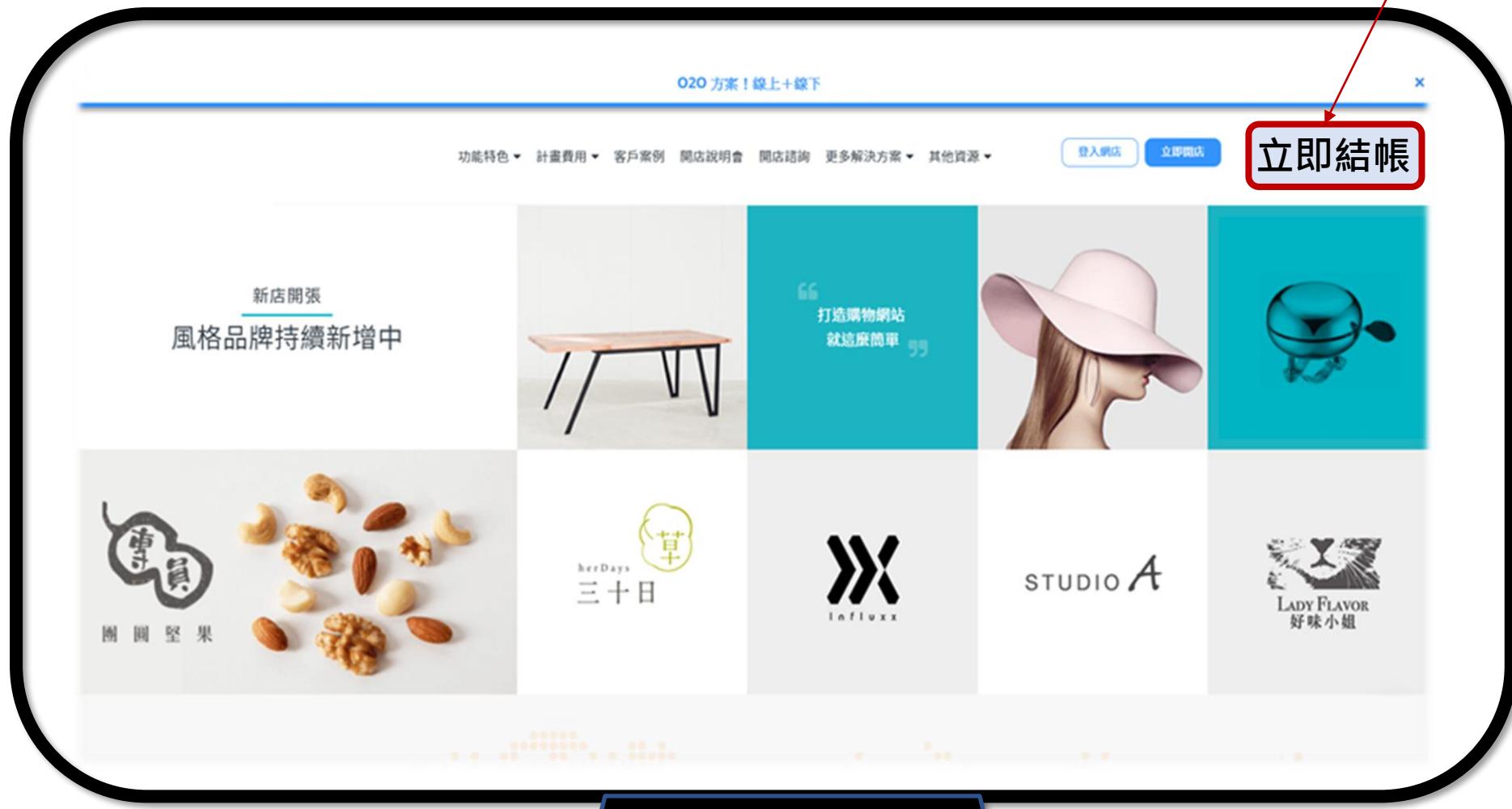
誰？

怎樣？

什麼？

**案例：**產品負責人PO跟視覺設計師要求在畫面上加一個快速結帳的按鈕，因為店員(客戶)抱怨操作步驟太繁複，走到結帳的畫面要太多道手續。

**加一個快速結帳的按鈕**



**故事：**產品負責人PO跟視覺設計師要求在畫面上加一個快速結帳的按鈕，因為店員(客戶)抱怨操作步驟太繁複，走到結帳的畫面要太多道手續。

範例：『如果我問消費者想要什麼，他會告訴我，“**要一匹更快的馬！**』

請問：用戶真正的需求/目標是什麼？難道是一匹更快的馬嗎？其實不是，用戶真正想要的是 ，更快的馬只是用戶自己提出的解決方案。

## 故事：

產品負責人PO跟視覺設計師要求在畫面上加一個快速結帳的按鈕，因為店員(客戶)抱怨操作步驟太繁複，走到結帳的畫面要太多道手續。

範例：『如果我問消費者想要什麼，他會告訴我，“**要一匹更快的馬！**』

請問：用戶真正的需求/目標是什麼？難道是一匹更快的馬嗎？其實不是，用戶真正想要的是更快速的到達目的地，更快的馬只是用戶自己提出的解決方案。



• 目標



• 用戶/角色



• 方式

• 方法



# 重做一次的

1. 上次學到什麼?

知識

2. 我的主要路徑對了嗎?

評估

3. 需要採用替代方案嗎?

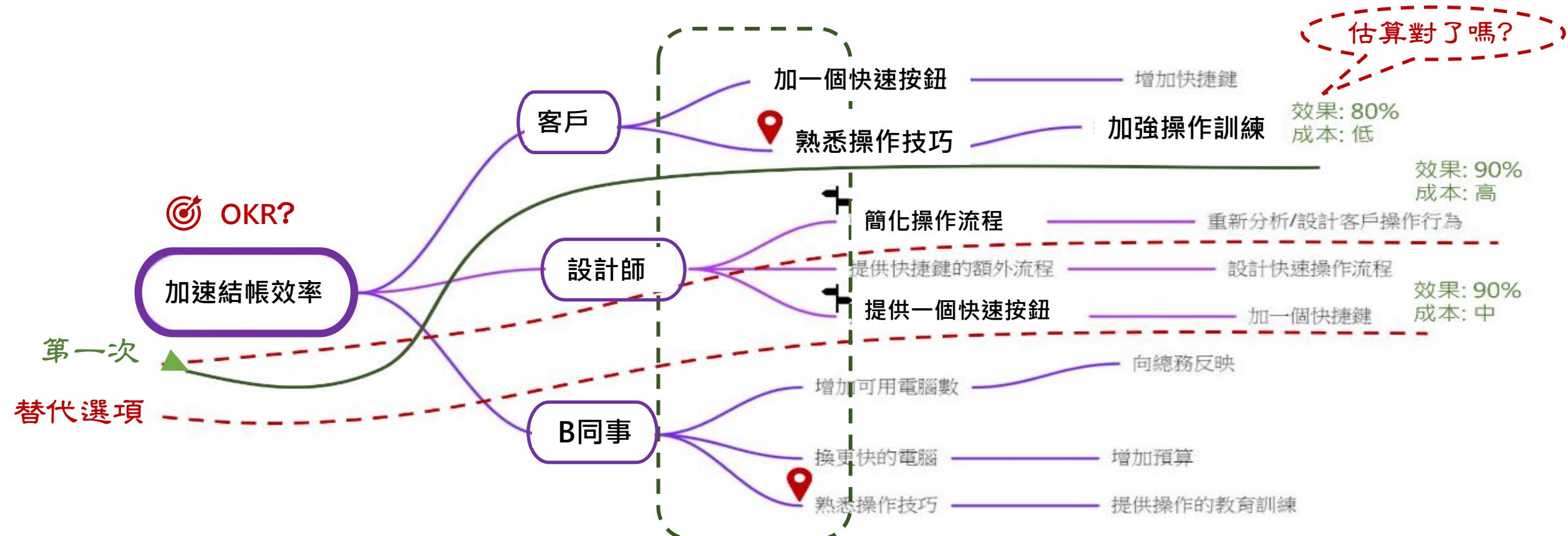
機會成本

Why ?

Who ?

How ?

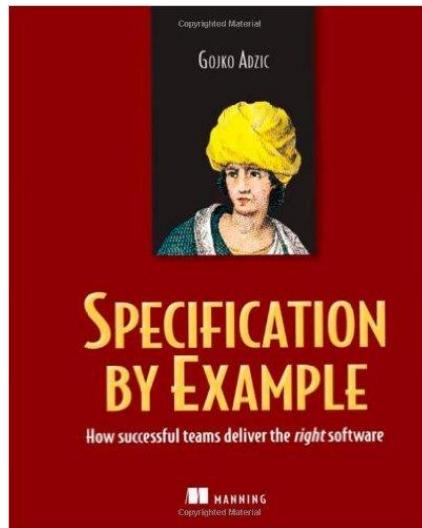
What ?



有興趣的話 ...

# 影響地圖

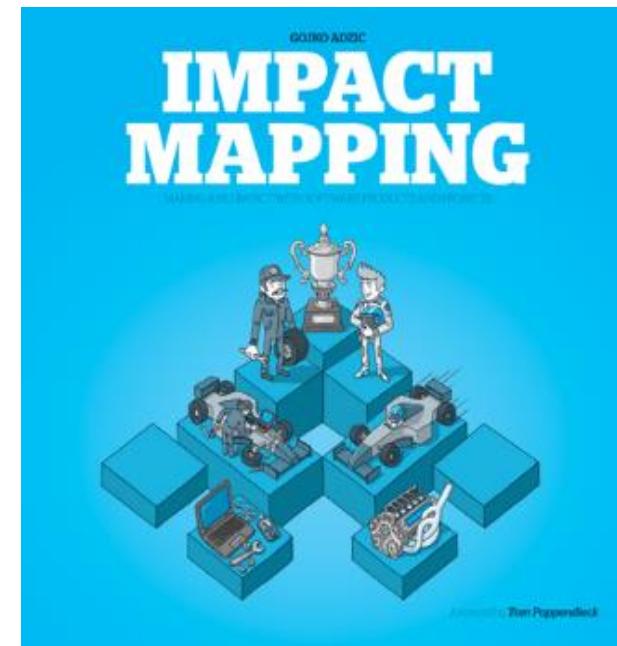
實例化需求 2012 JOLT



Gojko Adzic



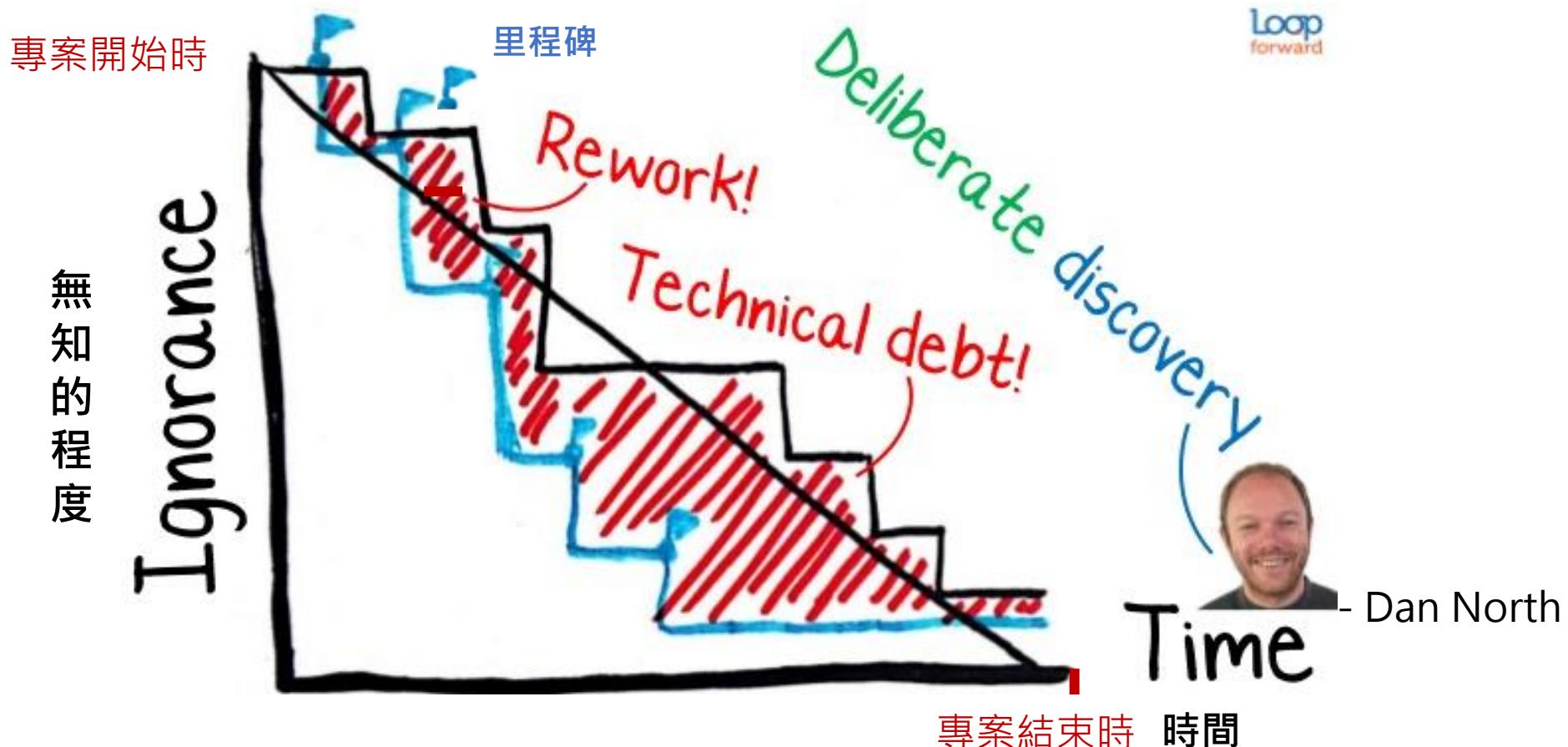
《影響地圖》



說明了如何把「**概念視覺化**」，提供一種 **看見需求的能力**，透過簡單的問自己四個問題: Why-Who-How-What 的分析方法，將策略以結構化的方式顯示出來。

# 無知是多元的…

Ignorance is multivariate



》 技術債 Technical debt: 是已知卻不做(已知的無知)你已經知道了，卻不去做。

# 無知是多元的…

Ignorance is multivariate



## 什麼樣的無知正在拖慢你的速度？

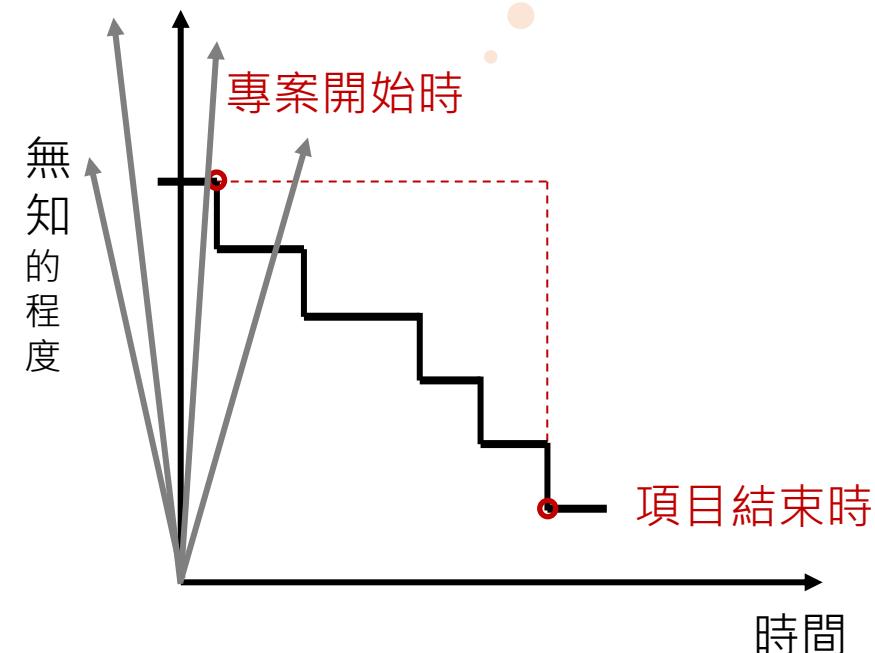
? 無知

你可能在某些項目上，有著某種程度上的無知 ...

- 你對相關領域的瞭解 (domain know how)
- 問題的本質
- 你的現有技術
- 其他可能的技術
- 組織上的約束
- 與利益相關者的關係...

》 你可能都不知道！

無知是多維度的





天真的以為 ...

## 通常我們會這麼想 ...

- 這次我們會更好地瞭解 ...
- 這次會有所不同
- 這次我們會準時完成的

這次需要多久呢?



通常我們會這麼想 ...

- 這次我們會更好地瞭解 ...
- 這次會有所不同
- 這次我們會準時完成的

這次需要多久呢?

→ 這次將與其他次；完全一樣！

無知比知識更易造就自信 - 達爾文: 意思是當你沒有足夠的知識來評估情勢，就很容易高估自己的能力。



通常我們會這麼想 ...

- 這次我們會更好
- 這次會有所改
- 這次我們會準時完

有人這麼做過嗎？

這次需要多久呢？

重做一次 ...

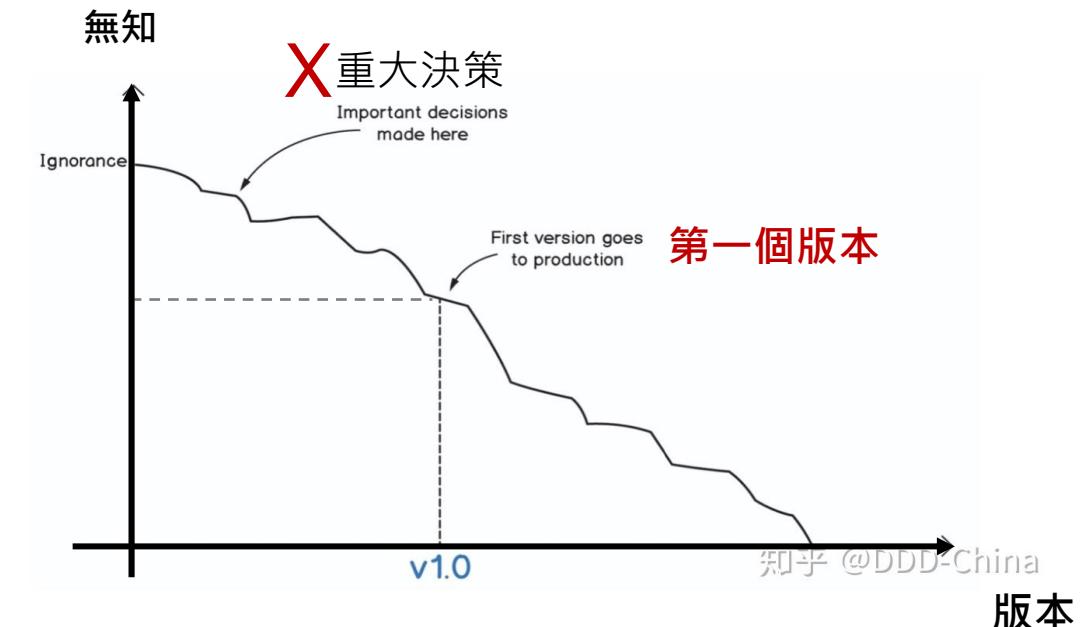
→ 結果答案在  $\frac{1}{4}$  到  $\frac{1}{2}$  的重複項目的時間並沒有減少。

- 莉茲·基奧 Liz Keogh

<https://lizkeogh.com/>

# 無知的無知是天生的 ...

don't know what I don't know is a given.



- 《領域驅動設計15年》 - By Alexey Zimarev

阿列克謝·日馬列夫

# 偏見是無知的孩子

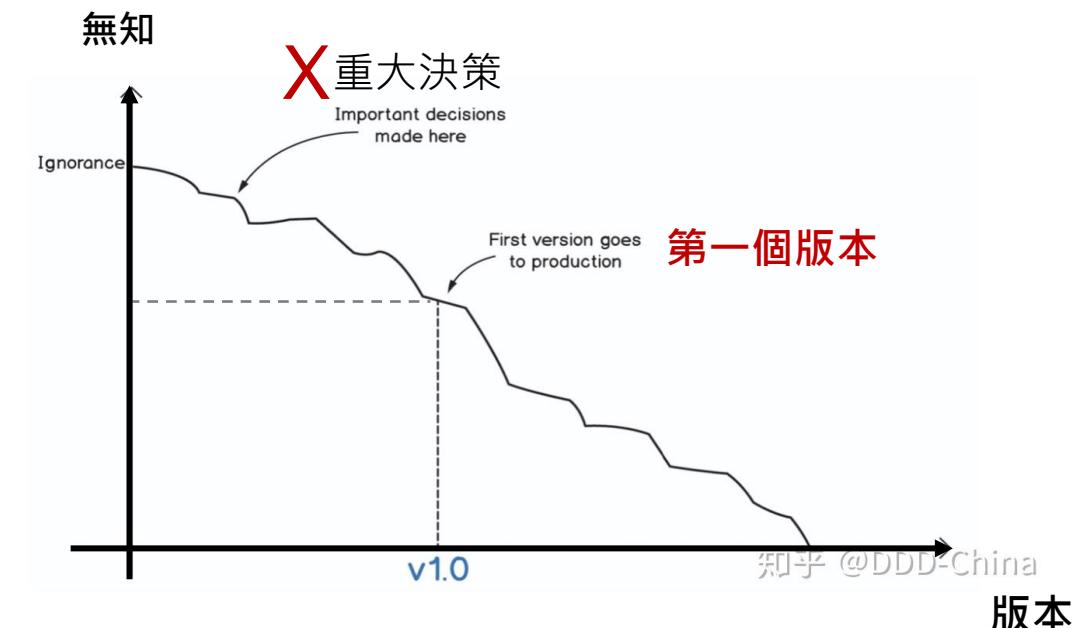
Prejudice is the child of ignorance

- 哈茲立特《拿破崙傳》

## 應對無知：

- ▶ 假設您始終無知，
- ▶ 假設某些特定的對流程的無知是您當前的約束，
- ▶ 通過積極解決無知來提高產出效能

解決已知的無知，是最大的效能。



- 《領域驅動設計15年》 - By Alexey Zimarev

阿列克謝·日馬列夫

# 偏見是無知的孩子

Prejudice is the child of ignorance

- 哈茲立特《拿破崙傳》

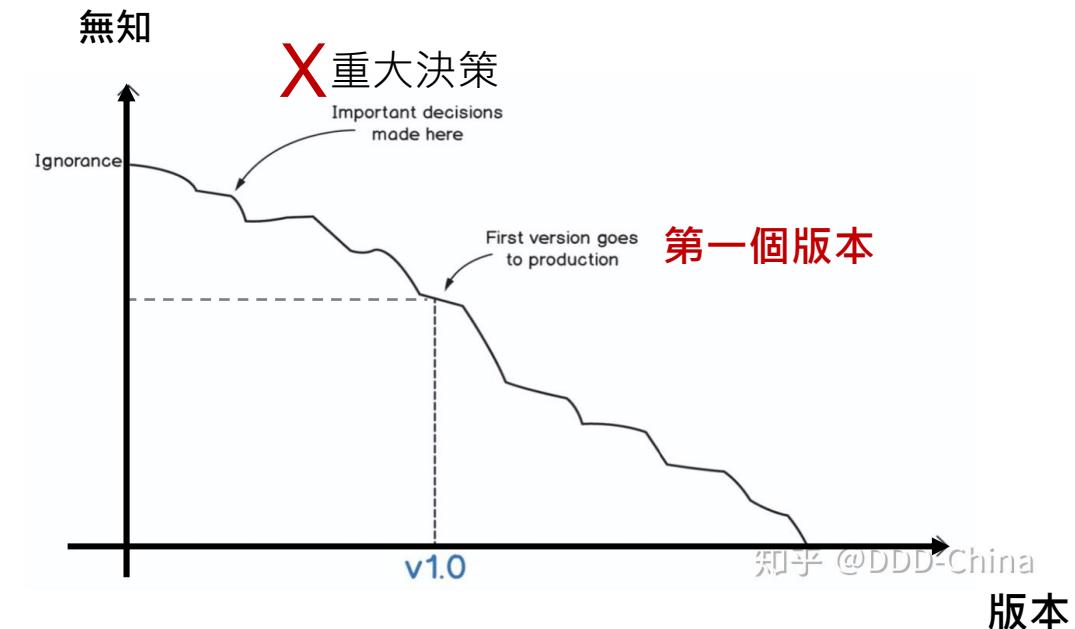
## 應對無知：

假設您始終無知，  
假設某些特定的對流程的無知是您當前的約束，  
通過積極解決無知來提高產出效能

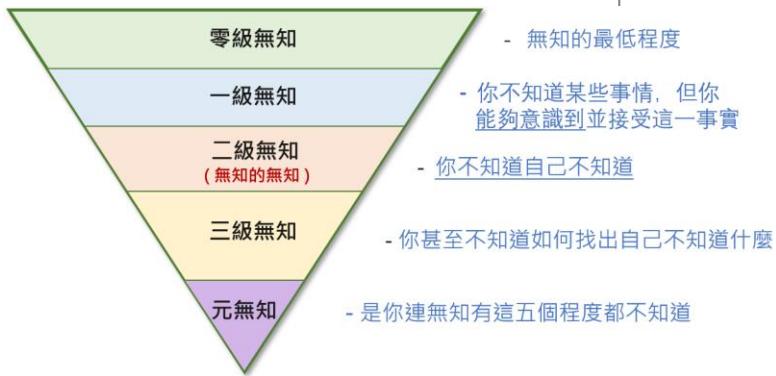
**解決已知的無知，是最大的效能。**

要考慮以下三個風險：

- 專案期間會存在一些不可預知的不利因素。
- 由於不可預知，根本無法提前知曉這些因素到底是什麼。
- 更糟的是，這些因素將對專案產生負面影響。



- 《領域驅動設計15年》 - By Alexey Zimarev  
阿列克謝·日馬列夫



## The 5 orders of ignorance

### 0OI = Lack of Ignorance

I (probably) know something. I have the answer and can explain it.

### 1OI = Lack of Knowledge

I don't know something, but I have a well formed question.

### 2OI = Lack of Awareness

I don't know that I don't know something. I have no idea what the right question is.

### 3OI = Lack of Process

I don't know a suitable way to find out I don't know that I don't know something.

### 4OI = Meta-ignorance

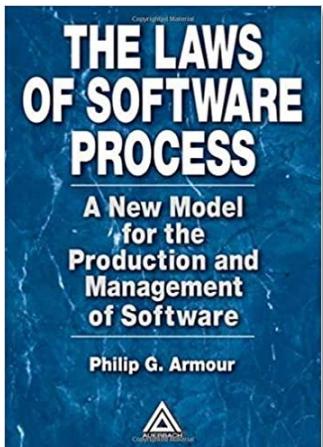
I have no clue how to find the right idea's to search for the right question.

### 4OI = Meta-ignorance

I don't know about the Five Orders of Ignorance.

# 《無知的五個程度》

將軟體開發看作是 知識獲取 和 減少無知 的過程



2003

*Phillip G. Armour*

## The Five Orders of Ignorance

*Viewing software development as knowledge acquisition and ignorance reduction.*

In my first column (Aug. 2000, p. 19), I argued that software is not a product, but rather a medium for the storage of knowledge. In fact, it is the fifth such medium that has existed since the beginning of time. The other knowledge storage media being, in historical order: DNA, brains, hardware and books. The reason software has become the storage medium of choice is that knowledge in software has been made *active*. It has escaped the confinement and volatility of knowledge in brains; it avoids the passivity of knowledge in books; it has the flexibility and speed of change missing from knowledge in DNA or hardware.

If software is not a product, then what is the product of our efforts to produce software? It is the knowledge contained in the software. It's rather easy to produce software that works, because we have to understand the meaning of "works." It's easy to produce simple

software because it doesn't contain much knowledge. Software is easier to produce using an application generator, because much of the knowledge is already stored in the application generator. Software is easy to produce if I've already produced this type of system before, because I have already obtained the necessary knowledge.

### Hacking

It is quite easy to show that software development is a knowledge-acquisition activity using a slightly exaggerated example. Imagine a hacking project. With hacking there is no real attempt to acquire the knowledge first, the project just hacks code. As the code is written and executed (testing may be too strong a word), there comes a point where validity of the knowledge in the code is checked somehow. This accomplishes two things: it identifies what in the code is "correct" (the knowledge) and what in the code is "incorrect" (what I call "unknowledge"). This unknownledge is often—somewhat incorrectly—considered to be defects. From a knowledge perspective, "unknowledge" is still knowledge; it just doesn't



- 2000年 · Philip G. Armor

- 2000年，Philip Armor

## 《無知的五個程度》

軟體不是產品，將軟體發展看作知識獲取和減少無知的過程。

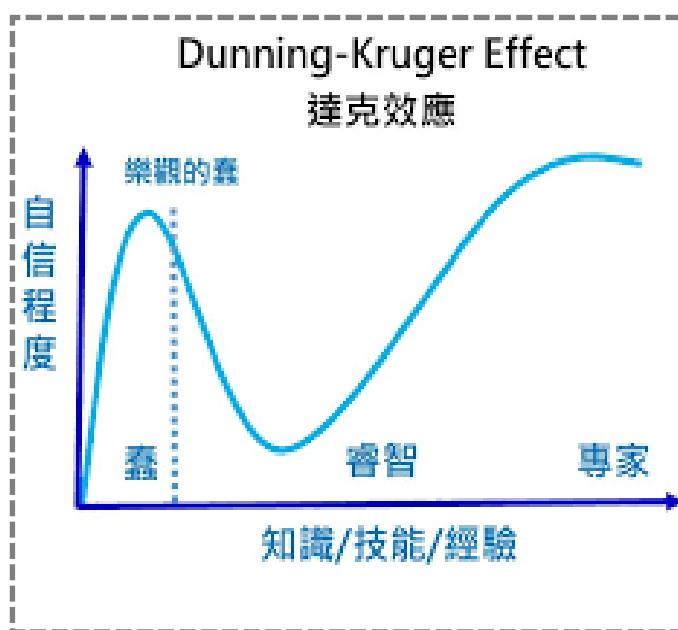
# 無知的分級

Orders Of Ignorance

## 無知的無知

Dunning-Kruger Effect

達克效應



無知比知識更易造就自信

零級無知

一級無知

二級無知  
(無知的無知)

三級無知

元無知

- 無知的最低程度

- 你不知道某些事情，但你能夠意識到並接受這一事實

- 你不知道自己不知道

- 你甚至不知道如何找出自己不知道什麼

- 是你連無知有這五個程度都不知道

# 無知的分級

Orders Of Ignorance

參考：

<https://wiki.c2.com/?OrdersOfIgnorance>

## 1. 零級無知 - 無知的最低程度

在這個層次上, 你無所不知, 因為掌握了大部分知識, 知道該做什麼, 及如何做。

## 2. 一級無知 - 你不知道某些事情，但你能夠意識到並接受這一事實

你希望獲得更多知識並將無知程度降低到零級, 因此你可以獲得掌握知識的管道。

## 3. 二級無知 - 你不知道自己不知道 無知的無知

最常見的情況是, 你僅僅得到一個解決方案的規格說明, 但其中並沒有描述這個解決方案要解決的問題是什麼。這種無知程度的另一種表現是, 自以為具有某種本來不具備的能力, 而根本沒有意識到這一點。這種人可能同時缺乏業務和技術知識, 在這個無知水準上會做出許多錯誤的決定。

## 4. 三級無知 - 你甚至不知道如何找出自己不知道什麼。

在這個層次上做任何事都很難, 因為顯然沒法和客戶溝通。從而甚至無法確認是否理解了客戶的問題, 從而達到二級水準。在這種情況下, 構建系統可能是唯一的選擇, 因為這是獲得反饋的唯一方法。

## 5. 最後一級 - “元無知”，是你連無知的五個程度都不知道。

# 無知的無知

I don't know that I don't know.

## 3. 二級無知

你不知道自己不知道

← 無知的無知

最常見的情況是，你僅僅得到一個解決方案的規格說明，但其中並沒有描述這個解決方案要解決的問題是什麼。這種無知程度的另一種表現是，自以為具有某種本來不具備的能力，而根本沒有意識到這一點。這種人可能同時缺乏業務和技術知識，在這個無知水準上會做出許多錯誤的決定。

# 無知的無知

I don't know that I don't know.

打破這種無知的方法 -----> 問 WHY?

## 3. 二級無知

你不知道自己不知道

← 無知的無知

最常見的情況是，你僅僅得到一個解決方案的規格說明，但其中並沒有描述這個解決方案要解決的問題是什麼。這種無知程度的另一種表現是，自以為具有某種本來不具備的能力，而根本沒有意識到這一點。這種人可能同時缺乏業務和技術知識，在這個無知水準上會做出許多錯誤的決定。

# 由無知的無知到一級無知

Orders Of Ignorance

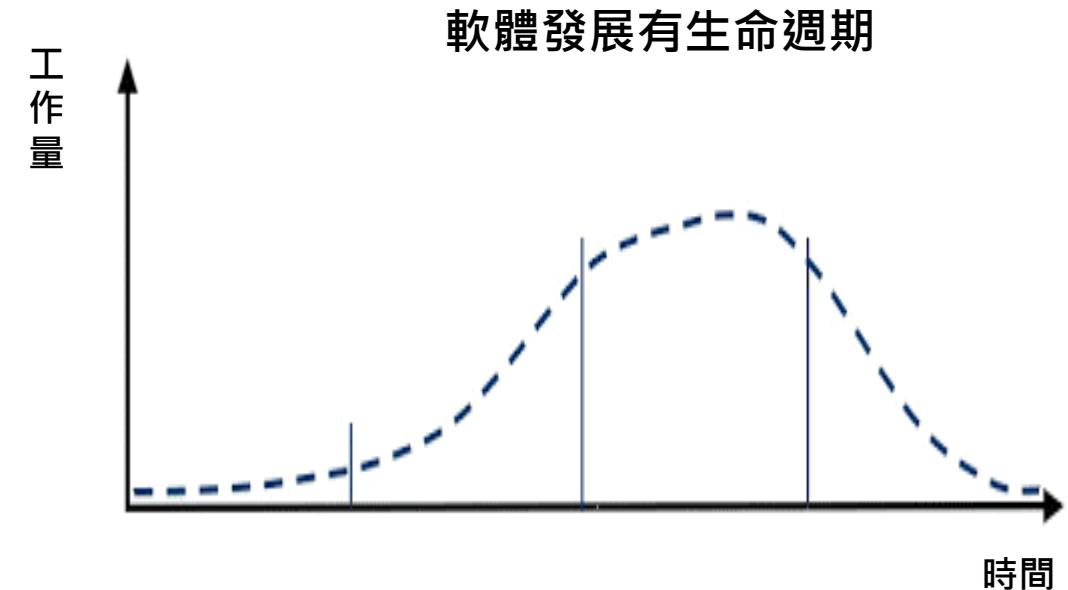
2. 一級無知 - 你不知道某些事情，但你 能夠意識到 並接受這一事實

你希望獲得更多知識並將無知程度降低到零級，因此你可以獲得掌握知識的管道。

3. 二級無知 - 你不知道自己不知道 ← 無知的無知

I don't know that I don't know.

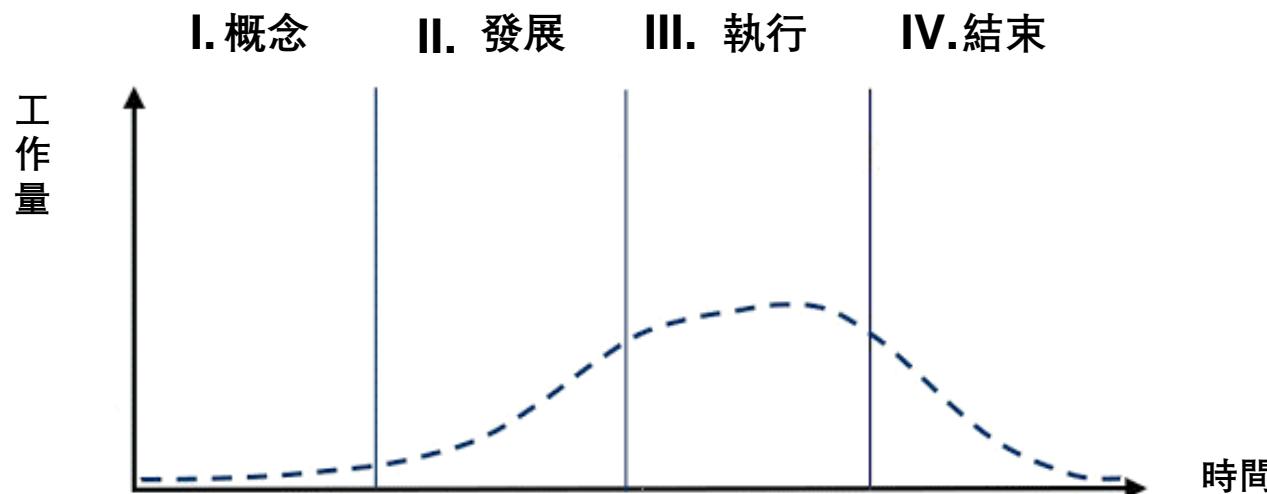
## 四、開發的本質 與刻意發現



# 軟體發展生命週期

從概念提出的那一刻開始，軟體產品就進入了軟體生命周期。

在經歷需求、分析、設計、實現、部署後，軟體將被使用併進入維護階段，直到最後由於缺少維護費用而逐漸消亡。這樣的一個過程，稱為“生命周期模型” Life Cycle Model。

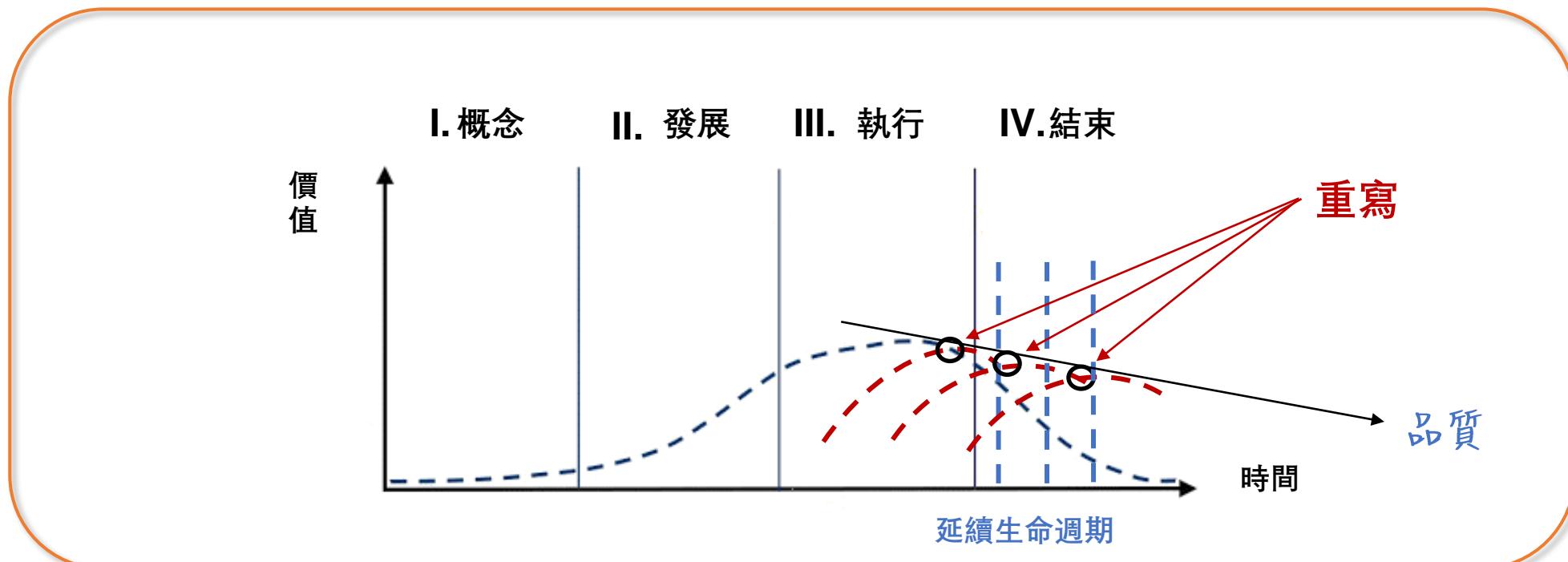


# 軟體發展有生命週期



重新開發還是重寫？

在延續產品生命週期的議題 Re-write 。

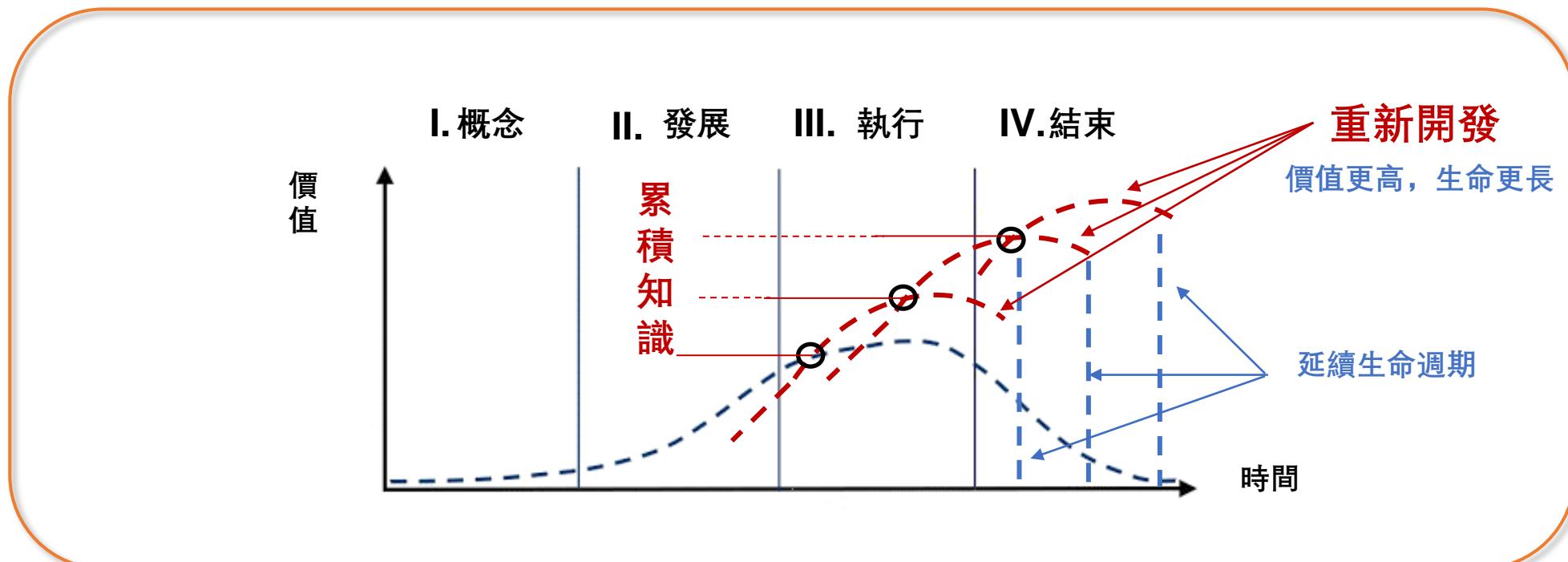


# 軟體發展有生命週期



重新開發還是重寫？

在延續產品生命週期的議題上 Re-development >> re-write 。



# 敏捷開發的迷失

## 開發的自然流程：

# 需求、分析、設計、實現、部署

敏捷開發為了避免過度的文檔：

## To Do – Doing - Code review ...



## 少了「**分析**」的步驟



# 敏捷團隊容易忽略 前期分析與文件

## 開發的自然流程：

# 需求、分析、設計、實現、部署

**敏捷開發為了避免過度的文檔：**

## To Do – Doing - Code review ...



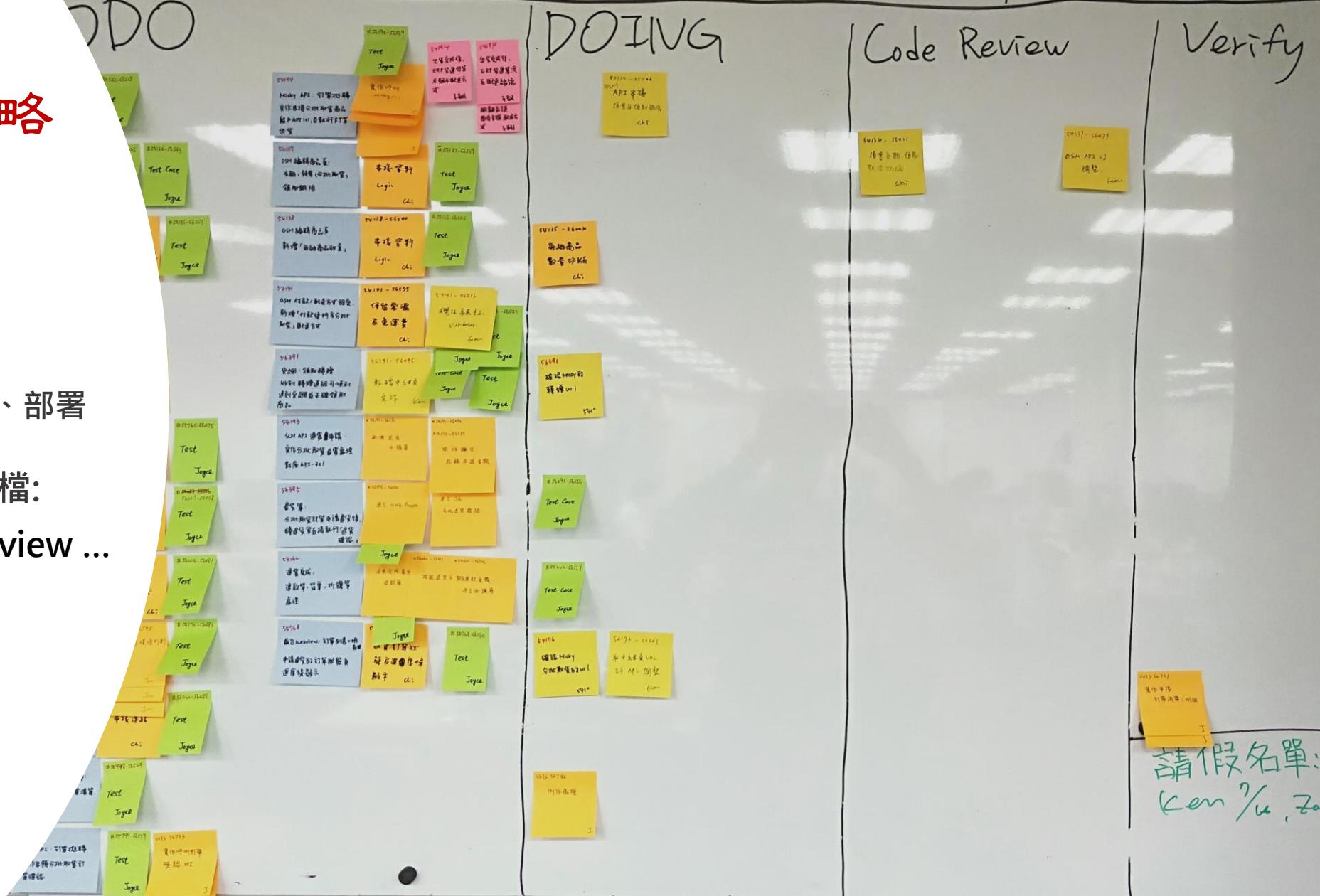
# 少了分析

少了可以減少「無知」的機會

## 少了充實開發知識的基礎

請支援收銀

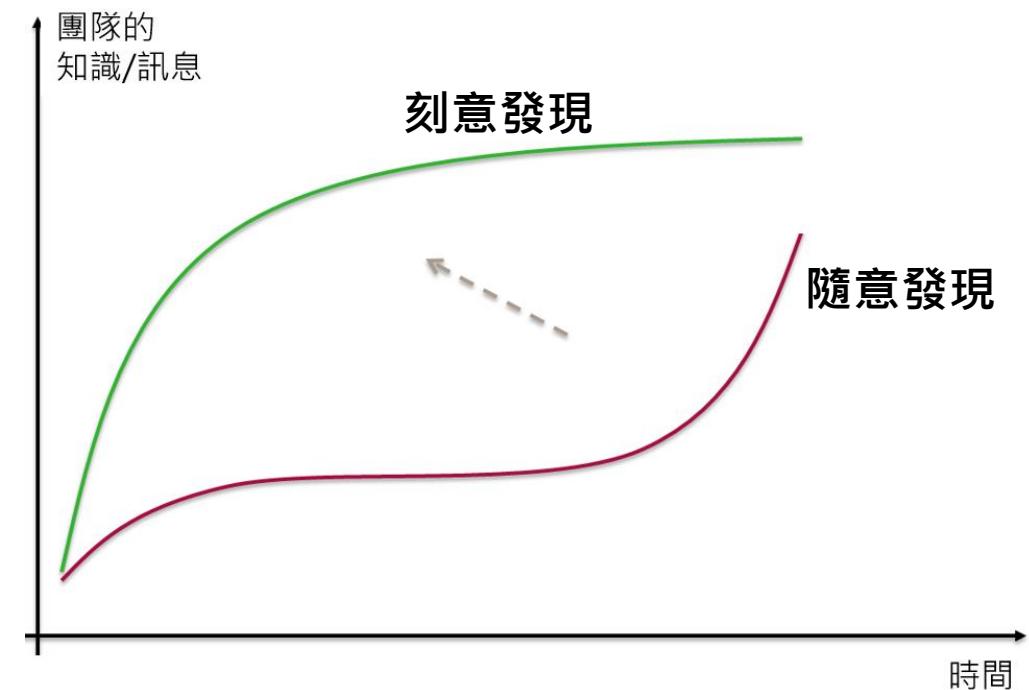
## Sprint 2 (7/ ~ 7/ )



## 刻意發現

Deliberate Discovery

- 撰寫程式、項目規畫、分析作業



# 參考資料

## 刻意發現介紹 – Dan North



Spike – 2<sup>nd</sup> Thought - Pair

自然開發 - 敏捷的迷失

不要怕過度分析，要前置學習

補充背景知識

- Dan North

# 刻意發現 用於撰寫程式上

Spike – 2<sup>nd</sup> Thought - Pair

穿測 – Pair programming

## Deliberate discovery in programming

### Spike-and-stabilise 穿測

- learn through evolving the “spike”
- choose to stabilise later – deferred, test-driven testing

### Design for the second case 替代方案

- you *might* gonna need it!

### Indirect discovery

### Travel in pairs 結伴同行

- optimise for learning: delivery will take care of itself

© Dan North, DRW

13

- Dan North

# 刻意發現 用於專案規畫上

遵循自然的邏輯思維方式

## Deliberate discovery in planning

Plan for at least some unexpected bad things 總是會出事的

Try natural planning (GTD) 遵循自然的邏輯思維方式

1. Purpose
2. Mission/vision/goals
3. Brainstorm 集思廣益
4. Organise
5. Next actions

Figure out your axes of ignorance. Then do it again.

*Beware the perils of fractal estimation*

# 刻意發現 用於分析作業上

前置分析

## Deliberate discovery in analysis

Don't fear "analysis paralysis" 不要害怕過度分析

- as long as it's reducing *relevant* ignorance

Ethnography 投資者在意的事

- What are your stakeholders caring about?
- What *should* they be caring about?

Play it forward 前置思維

- Who are you trying to reach?
- What do you want their experience to be?

*Understanding the domain is a whole team activity*

群策群力

© Dan North, DRW

12

- Dan North

# — Tester

專案重做一次，  
對 測試人員 的意義.



測試過程

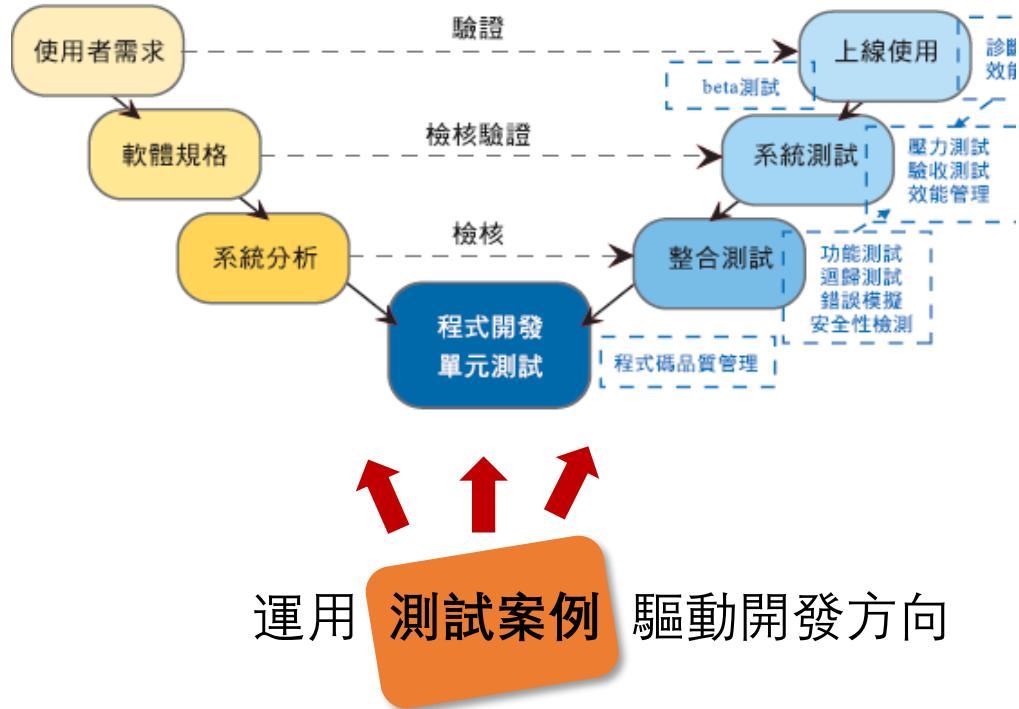


測試錦囊

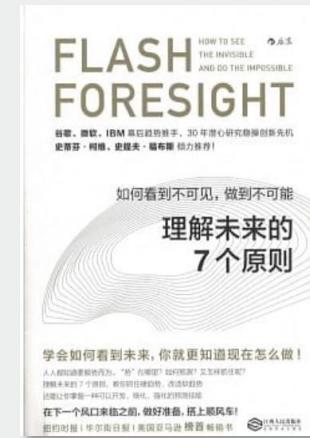
# Tester

專案重做一次，  
對 測試人員 的意義。

## 融入開發過程



# 《理解未來的七個原則》



- 一、從確定性開始：了解硬(必然)趨勢 / (非必然)軟趨勢。
- 二、洞察先機：先發制人，防微杜漸 - 風險思維。
- 三、變革：只有改變是不夠的，變革才是個人和企業面對未來的重要法則。
- 四、跳出你面臨的問題：跳出這個問題，找到其他的方法，做更好的決策。
- 五、反其道而行：這樣沒準可以找到更好的解決方法。
- 六、重新定義和再創造：未來，新的社會環境會要求更強的再創造能力。
- 七、主導未來：新的未來有三個特點：溝通、合作、信任。

► 這七個原則其實無法直接幫助我們預測到未來，但是它們可以構成了一種克服無知的思考模式，按照這種思考模式，在我們遇到問題時，便能夠根據未來社會的某些發展趨勢，找到解決問題的方法，從而理解未來，預測未來，主導未來。

將經驗轉換為能力



# 問題與回答

【問題】 【分析】 【回答】 【確認】 【完成】

5

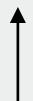
3

1

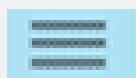
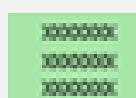
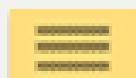
3

【完成】

High



Low



收集類似問題最多  
三個，一起回答。

每個問題回答二分鐘

提問者點頭認同回答

Thanks for your listening.

謝謝你的聆聽。



#填寫 活動回饋問卷，就可以取得第一手講師講義包唷！  
#填寫 職涯問卷，就可以向現場工作人員兌換 91APP 專屬小禮物唷

稍待片刻，待會將帶來更精彩的議程  
提醒您，我們將於 15:35 準時開始

# 專案開始之初，首重看見全貌

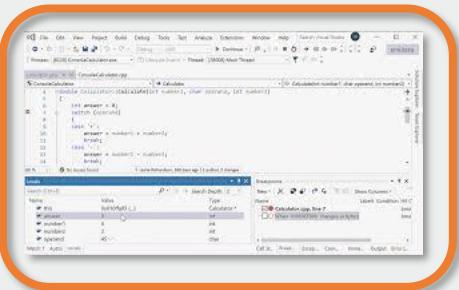
一旦；當你把眼光投注在哪一個要項的時候，  
實際上你就只看到那一部分，  
你的思緒將被那一部分的內容所牽動，很難再看見其他的事...  
所以我們要退後一步，

# 專案開始之初，首重看見全貌

一旦；當你把眼光投注在哪一個要項的時候，  
實際上你就只看到那一部分，  
你的思緒將被那一部分的內容所牽動，很難再看見其他的事...  
所以我們要退後一步，  
不！有時要退後很多步，才能比較清晰地看見全貌。

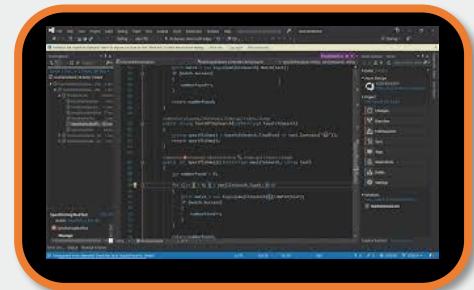
- 請思考你的「退後一步」是什麼？

# 軟體工程師作抉擇時的思維方式



白底黑字

退後一步：機會成本  
opportunity cost



黑底白字

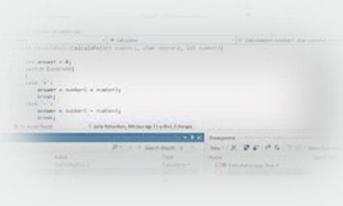


```
33     self.fingerprints = set()
34     self.logduplicates = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, 'fingerprint.log'), 'w')
39         self.file.seek(0)
40         self.fingerprints.update(self._read())
41
42     @classmethod
43     def from_settings(cls, settings):
44         debug = settings.getbool('superuser_log_level')
45         return cls(job_dir(settings), debug)
46
47     def request_seen(self, request):
48         fp = self.request_fingerprint(request)
49         if fp in self.fingerprints:
50             return True
51         self.fingerprints.add(fp)
52         if self.file:
53             self.file.write(fp + os.linesep)
54
55     def request_fingerprint(self, request):
56         pass
```

# 軟體工程師作抉擇時的思維方式

退後一步：機會成本

**opportunity cost**



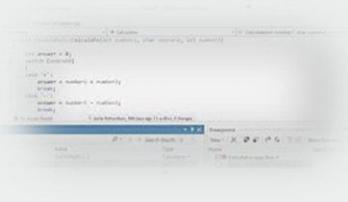
每一個決定都是一次權衡.

- Dan North

# 軟體工程師作抉擇時的思維方式

退後一步：機會成本

opportunity cost



每一個決定都是一次權衡。

- Dan North

面對技術債；如果再給你一次機會，你還會這麼做嗎？

問個問題，暖身一下…

敏捷的素養與寫程式寫得好壞  
有關係嗎？

# 敏捷的素養與寫程式寫得好壞

有關係嗎？

寫程式的素養



正確的構建  
Building the thing right

速度、品質

# 敏捷的素養與寫程式寫得好壞

有關係嗎？

敏捷的素養

正確的產品  
Building the right thing

方向

寫程式的素養

正確的構建  
Building the thing right

速度、品質



# 你將會聽到的...

- ✓ • 工程師的機會成本
  - 敏捷與寫程式的關係
  - 面對技術債
- 專案重做一次，會怎麼樣?
  - 限制理論的思維
- ★ - 策略規劃的影響地圖
  - 刻意發現
- 善用軟體生命週期
  - 累積知識的重要性

# 你將不會聽到的...

- 限制理論實用範例
- 影響地圖深入講解
- OKR ...



限制理論

Theory of Constraints

專案開發生命週期



## 專案重做的限制理論與刻意發現

無知與模糊

刻意發現

系統一定會有約束且約束會一直存在著



如何解決瓶頸？

限制理論

策略與無知

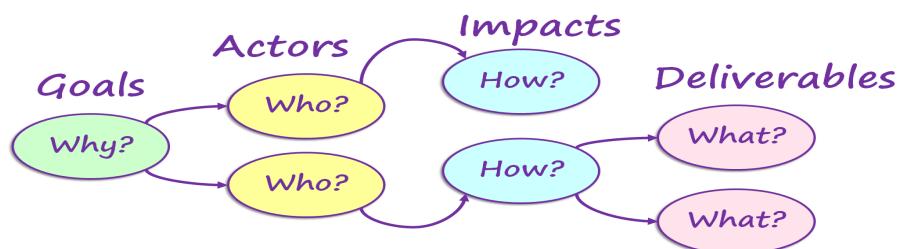
專案開發生命週期

刻意發現

## 限制理論

### Theory of Constraints

『無知』的分級



策略工具 - 影響地圖

## 專案開發生命週期

刻意發現

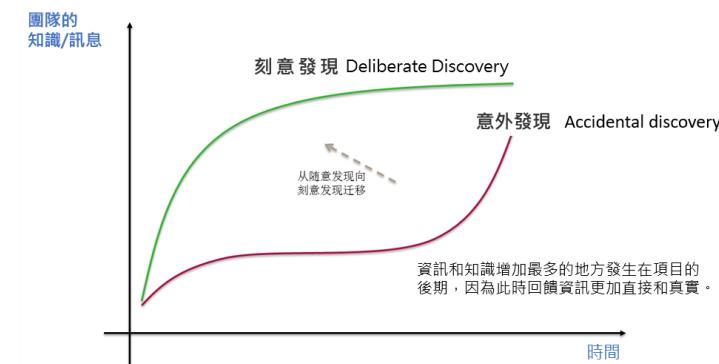
# 限制理論

## Theory of Constraints

無知與模糊

# 專案開發生命週期

刻意發現



如何刻意發現？

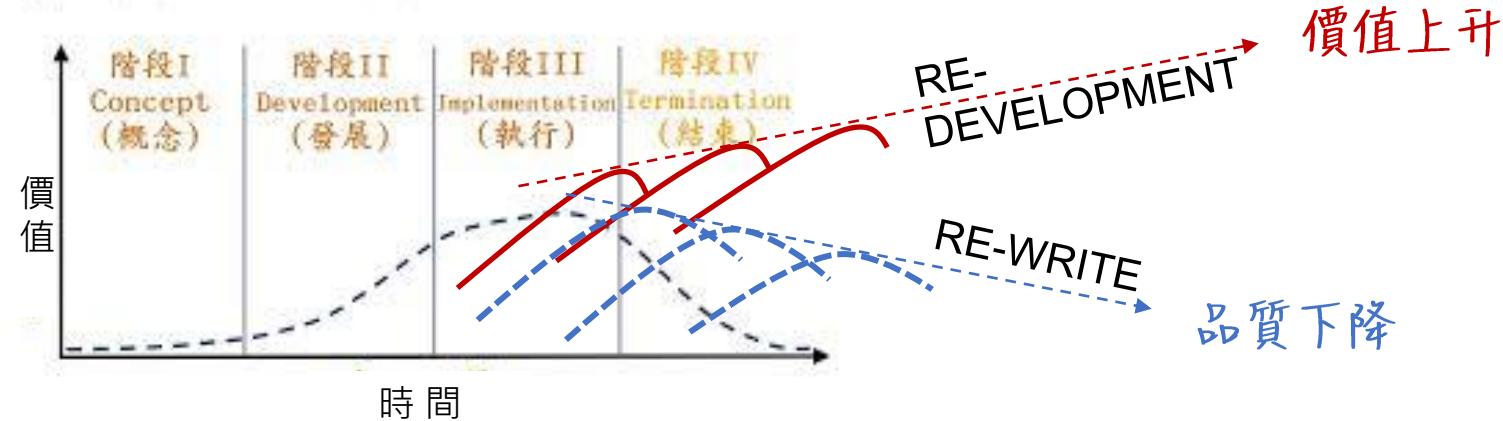
## 限制理論

## Theory of Constraints

無知與模糊

## 專案開發生命週期

專案生命週期的四個主要階段



刻意發現

系統一定會有約束且約束會一直存在著

價值上升

### 專案重做的意義

具體知識的累積  
機會成本的考量



無知與模糊 - 影響地圖

如何刻意發現？