VIB

Code Complete

Mục lục

VIB

1. Welcome to Software Construction

1.1. What Is Software Construction?

- Xây dựng phần mền chủ yếu là việc viết mã và gỡ lỗi, nhưng cũng bao gồm thiết kế chi tiết, lập kế hoạch xây dựng, kiểm thử đơn vị, tích hợp, kiểm thử tích hợp và các hoạt động khác.
- Quy trình phát triển phần mềm gồm các bước sau:
    - Problem definition
    - Requirements development
    - Construction planning
    - Software architecture, or high-level design
    - Detailed design
    - Coding and debugging
    - Unit testing
    - Integration testing
    - Integration
    - System testing
    - Corrective maintenance
- Nhiệm vụ thụ thể để xây dựng phần mềm:
    - Verifying that the groundwork has been laid so that construction can proceed successfully
    - Determining how your code will be tested
    - Designing and writing classes and routines
    - Creating and naming variables and named constants
    - Selecting control structures and organizing blocks of statements
    - Unit testing, integration testing, and debugging your own code
    - Reviewing other team members' low-level designs and code and having them
    - review yours
    - Polishing code by carefully formatting and commenting it
    - Integrating software components that were created separately
    - Tuning code to make it faster and use fewer resources

1.2. Why Is Software Construction Important?

- Dự án phần mềm lý tưởng trải qua quá trình phát triển yêu cầu cẩn thận và thiết kế kiến trúc trước khi xây dựng bắt đầu. Dự án lý tưởng trải qua quá trình kiểm thử hệ thống toàn diện được kiểm soát thống kê sau khi xây dựng hoàn thành. Tuy nhiên, những dự án thực tế không hoàn hảo thường bỏ qua yêu cầu và thiết kế để nhảy vào giai đoạn xây dựng. Họ bỏ kiểm thử vì có quá nhiều lỗi cần sửa và họ đã hết thời gian. Nhưng cho dù dự án được thực hiện vội vã hay lên kế hoạch kém, bạn không thể bỏ qua giai đoạn xây dựng. Vì giai đoạn xây dựng thường chiếm từ 30 đến 80 phần trăm thời gian tổng cộng đã dành cho dự án. Bất cứ điều gì chiếm nhiều thời gian dự án như vậy đều ảnh hưởng đến sự thành công của dự án.

1.3. Summary of main content in this chapter

- Xây dựng phần mềm là hoạt động trung tâm trong quá trình phát triển phần mềm; xây dựng là hoạt động duy nhất được đảm bảo xảy ra trong mọi dự án.
- Các hoạt động chính trong giai đoạn xây dựng bao gồm thiết kế chi tiết, viết mã, gỡ lỗi, tích hợp và kiểm thử của nhà phát triển (unit testing and integration testing).
- Các thuật ngữ phổ biến khác để chỉ về software construction là "coding" và "programming."
- Chất lượng của giai đoạn xây dựng có ảnh hưởng đáng kể đến chất lượng của phần mềm.

2. Metaphors for a Richer Understanding of Software Development

2.1. The Importance of Metaphors

- Thực tế cho thấy, phép ẩn dụ là một cách tiếp cận và trừu tượng hóa các khái niệm, cho phép tư duy của người ta hoạt động ở một mức cao hơn và tránh được các sai lầm cấp thấp. Trong phần mền, cũng tương tự trong thực tế, giúp ta có cái hình đa chiều và tư duy mỡ đổi mới, để từ phép ẩn dụ tệ hơn sang phép ẩn dụ tốt hơn hay từ khái quát hơn sang bao quát hơn.

2.2. How to Use Software Metaphors

- Sử dụng phép ẩn dụ để giúp hiểu sâu hơn về các vấn đề và quy trình lập trình. Sử dụng chúng để giúp suy nghĩ về các hoạt động lập trình của mình và để giúp tưởng tượng ra cách thực hiện tốt hơn. Ta sẽ không thể nhìn vào một dòng mã và nói rằng nó vi phạm một trong những phép ẩn dụ mô tả trong chương này. Tuy nhiên, theo thời gian, người sử dụng phép ẩn dụ để làm sáng tỏ quá trình phát triển phần mềm sẽ được nhận xét là người hiểu biết về lập trình hơn và tạo ra mã tốt hơn một cách nhanh chóng hơn so với những người không sử dụng chúng.

2.3. Common Software Metaphors

- **Software Penmanship: Writing Code**: là một quá trình phát triển phần mềm quá đơn giản và cứng nhắc để có thể làm việc hiệu quả. Thường được sử dụng cho dự án cá nhân hoặc dự án có quy mô nhỏ.
- **Software Farming: Growing a System**: là quá trình thiết kế một phần theo ý tưởng viết mã một phần, kiểm thử một phần và thêm nó vào hệ thống từng chút một. Bằng cách tiến hành từng bước nhỏ, giúp giảm thiểu rủi ro mà bạn có thể gặp phải tại bất kỳ lúc nào.
- **Software Oyster Farming: System Accretion**: Khi xây dựng hệ thống chỉ cần xây dựng lớp giả cho từng chức năng cơ bản mà đã được xác định từ trước sau đó từ từ thêm code hoàn chỉnh vào từng chức năng cụ thể cuối cùng sẽ tạo ra được chương trình hoàn chỉnh.
- **Software Construction: Building Software**: Xây dựng phần mềm ám chỉ các giai đoạn khác nhau của kế hoạch, chuẩn bị và thực hiện có tính chất và mức độ khác nhau tùy thuộc vào cái đang được xây dựng. Lên kế hoạch cẩn thận không nhất thiết phải là kế hoạch chi tiết hoặc quá nhiều kế hoạch. Phép ẩn dụ về xây dựng cung cấp cái nhìn sâu sắc vào các dự án phần mềm cực kỳ lớn. Và bởi vì hậu quả của việc thất bại trong một cấu trúc cực kỳ lớn là nghiêm trọng, cấu trúc

phải được thiết kế vượt qua mức cần thiết. Người xây dựng tạo và kiểm tra kế hoạch một cách cẩn thận. Xây dựng với khoảng dự phòng an toàn; Ví dụ khi xây dựng một tòa nhà tốt hơn là trả thêm 10% để có vật liệu mạnh hơn, thay vì để một tòa nhà chọc trời đổ sập.

- **Applying Software Techniques: The Intellectual Toolbox**: Tùy vào từng trường hợp, công việc mà sẽ sữ dụng kỹ thuật, thủ nhật, quy tắc, công cụ phù hợp để mang lại hiệu quả cao nhất.
- **Combining Metaphors**: Vì phép ẩn dụ là những phương pháp học hơn là thuật toán, chúng không loại trừ lẫn nhau. Bạn có thể sử dụng cả phép ẩn dụ về tích luỹ và phép ẩn dụ về xây dựng. Nên có thể sữ dụng kết hợp.

2.4. Summary of main content in this chapter

- Phép ẩn dụ là các phương pháp học, không phải thuật toán. Do đó, chúng thường hơi mơ hồ.
- Phép ẩn dụ giúp bạn hiểu quá trình phát triển phần mềm bằng cách liên kết nó với các hoạt động bạn đã biết.
- Một số phép ẩn dụ tốt hơn các phép ẩn dụ khác.
- Coi xây dựng phần mềm giống với xây dựng công trình gợi ý rằng cần chuẩn bị cẩn thận và làm sáng tỏ sự khác biệt giữa các dự án lớn và nhỏ.
- Suy nghĩ về các thực hành phát triển phần mềm như là các công cụ trong một hộp công cụ trí tuệ cũng gợi ý rằng mỗi lập trình viên có nhiều công cụ và không có một công cụ duy nhất phù hợp cho mọi công việc. Lựa chọn công cụ phù hợp cho mỗi vấn đề là một trong những yếu tố quan trọng để trở thành một lập trình viên hiệu quả.
- Các phép ẩn dụ không loại trừ nhau. Hãy sử dụng sự kết hợp của các phép ẩn dụ hoạt động tốt nhất cho bạn.

3. Measure Twice, Cut Once:Upstream Prerequisites

3.1. Importance of Prerequisites

- Khi tập trung vào phần nào nhiều hơn trong quá trình phát triển phần mền thì bạn sẽ tập trung vào yêu cầu đầu ra của phần đó nhiều hơn, đó là việc xác định điều kiện tiên quyết.
- Ví dụ khi bạn chọn phần đầu của quá trình phát triển làm trọng tâm thì sẽ tập trung vào lên kế hoạch, yêu cầu, thiết kế. Nếu tập trung vào phần giữa thì sẽ dành thời gian để xây dựng phần mền. Còn nếu tập trung vào phần cuối thì bạn đang nhấn mạnh tính chất lượng của chương trình và đời hỏi tập trung vào kiểm thử nhiều hơn.
- Nên việc chọn các điều kiện tiên quyết rất quan trọng và nên dành thời gian để thực hiện các điều kiện tiên quyết trước khi bắt đầu xây dựng phần mền
- Mục tiêu chung của chuẩn bị là giảm thiểu rủi ro: một người lập kế hoạch dự án tốt loại bỏ các rủi ro chính ngay từ đầu để phần lớn dự án có thể diễn ra một cách suôn sẻ nhất có thể. Rủi ro dự án phổ biến nhất trong phát triển phần mềm là yêu cầu kém và kế hoạch dự án kém, do đó chuẩn bị thường tập trung vào việc cải thiện yêu cầu và kế hoạch dự án.
- Suy nghĩ về cách xây dựng hệ thống trước khi bắt đầu vào xây dựng chúng.

3.2. Determine the Kind of Software You're Working On

- Các loại dự án phần mềm khác nhau đòi hỏi cân bằng khác nhau giữa việc chuẩn bị và xây dựng. Mỗi dự án đều là duy nhất, nhưng dự án thường rơi vào các phong cách phát triển chung. Bảng bên dưới hiển thị ba loại dự án phổ biến nhất và liệt kê các thực hành thường phù hợp nhất với mỗi loại dự án.

	Kind of Software		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Typical applications	Internet site Intranet site Inventory management Games Management information systems Payroll system	Embedded software Games Internet site Packaged software Software tools Web services	Avionics software Embedded software Medical devices Operating systems Packaged software
Life-cycle models	Agile development (Extreme Programming, Scrum, timebox development, and so on) Evolutionary prototyping	Staged delivery Evolutionary delivery Spiral development	Staged delivery Spiral development Evolutionary delivery
Planning and management	Incremental project planning As-needed test and QA planning Informal change control	Basic up-front planning Basic test planning As-needed QA planning Formal change control	Extensive up-front planning Extensive test planning Extensive QA planning Rigorous change control
Requirements	Informal requirements specification	Semiformal requirements specification As-needed requirements reviews	Formal requirements specification Formal requirements inspections
Design	Design and coding are combined	Architectural design Informal detailed design As-needed design reviews	Architectural design Formal architecture inspections Formal detailed design Formal detailed design inspections
Construction	Pair programming or individual coding Informal check-in procedure or no check-in procedure	Pair programming or individual coding Informal check-in procedure As-needed code reviews	Pair programming or individual coding Formal check-in procedure Formal code inspections
Testing and QA	Developers test their own code Test-first development Little or no testing by a separate test group	Developers test their own code Test-first development Separate testing group	Developers test their own code Test-first development Separate testing group Separate QA group

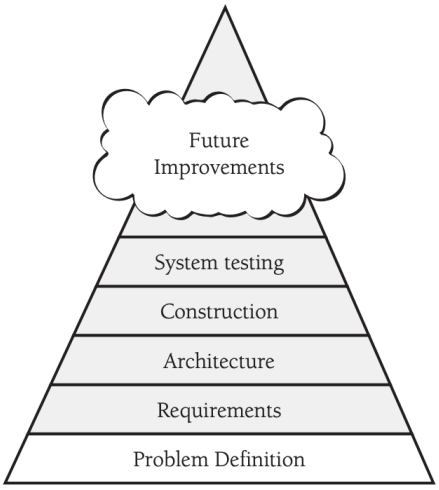
- Tập trung vào các tiên quyết có thể giảm chi phí bất kể bạn sử dụng phương pháp lặp lại hay tuần tự. Phương pháp lặp lại thường là một lựa chọn tốt vì nhiều lý do, nhưng một phương pháp lặp lại mà bỏ qua các tiên quyết có thể dẫn đến việc tăng chi phí đáng kể hơn so với một dự án tuần tự mà chú ý đến các tiên quyết.
- Hầu hết các dự án không phải hoàn toàn tuần tự cũng không hoàn toàn lặp đi lặp lại. Không thể thực hiện việc chỉ định 100% yêu cầu hoặc thiết kế từ đầu, nhưng hầu hết các dự án tìm thấy giá trị trong việc xác định ít nhất là các yêu cầu quan trọng nhất và các yếu tố kiến trúc sớm.
- Một nguyên tắc phổ biến là lập kế hoạch để chỉ định khoảng 80% yêu cầu từ đầu, dành thời gian để có thể chỉ định các yêu cầu bổ sung sau này, và sau đó thực hiện kiểm soát thay đổi hệ thống để chỉ chấp nhận những yêu cầu mới có giá trị nhất khi dự án tiến triển. Một phương án khác là chỉ định chỉ khoảng 20% yêu cầu quan trọng nhất từ đầu và lập kế hoạch để phát triển phần còn lại của phần mềm theo các bước nhỏ, chỉ định yêu cầu và thiết kế bổ sung khi tiến hành.

Lựa chọn giữa phương pháp tiếp cận lặp lại hay tuần tự:

- Bạn có thể chọn một phương pháp tuần tự trước khi:
    - Yêu cầu khá ổn định.
    - Thiết kế đơn giản và khá rõ ràng.
    - Đội phát triển quen thuộc với lĩnh vực ứng dụng.
    - Dự án không chứa nhiều rủi ro.
    - Dự báo dài hạn quan trọng.
    - Chi phí của việc thay đổi yêu cầu, thiết kế và mã nguồn ở bước sau có thể cao.
- Bạn có thể chọn một phương pháp lặp lại để tiến hành khi:
    - Yêu cầu không được hiểu rõ hoặc bạn dự đoán chúng sẽ không ổn định vì các lý do khác.
    - Thiết kế phức tạp, đầy thách thức hoặc cả hai.
    - Đội phát triển không quen thuộc với lĩnh vực ứng dụng.
    - Dự án chứa nhiều rủi ro.
    - Dự báo dài hạn không quan trọng.
    - Chi phí của việc thay đổi yêu cầu, thiết kế và mã nguồn ở bước sau có thể thấp.
- Phương pháp lặp lại thường hữu ích nhiều hơn so với phương pháp tuần tự. Bạn có thể điều chỉnh các tiên quyết cho dự án cụ thể của bạn bằng cách làm cho chúng cụ thể hơn hoặc ít hơn, hoặc hoàn toàn hoặc không hoàn toàn theo yêu cầu của bạn.

3.3. Problem-Definition Prerequisite

- Tiền đề đầu tiên bạn cần đáp ứng trước khi bắt đầu giai đoạn xây dựng là một tuyên bố rõ ràng về vấn đề mà hệ thống dự kiến giải quyết gọi là "định nghĩa vấn đề".



Hình 3-4: Định nghĩa vấn đề đặt nền móng cho phần còn lại của quá trình lập trình.

- Qua hình 3-4 có thể thấy Problem Definition diễn ra trước công việc yêu cầu chi tiết, đó là một cuộc điều tra sâu hơn về vấn đề.
- Nếu thiếu một định nghĩa vấn đề tốt, hậu quả là không xác định rõ vấn đề là bạn có thể lãng phí rất nhiều thời gian để giải quyết vấn đề sai. Điều này là một hình phạt kép vì bạn cũng không giải quyết đúng vấn đề.

3.4. Requirements Prerequisite

- Yêu cầu mô tả chi tiết về những gì mà một hệ thống phần mềm dự kiến phải thực hiện và chúng là bước đầu tiên hướng tới một giải pháp. Hoạt động về yêu cầu còn được gọi là "phát triển yêu cầu," "phân tích yêu cầu," "phân tích," "định nghĩa yêu cầu," "yêu cầu phần mềm," "đặc tả," "đặc tả chức năng," và "đặc tả."
- Một tập hợp yêu cầu rõ ràng có ý nghĩa quan trọng với một số lý do.
- Yêu cầu rõ ràng giúp đảm bảo rằng người dùng, thay vì lập trình viên, định hướng chức năng của hệ thống. Nếu yêu cầu rõ ràng, người dùng có thể xem xét và đồng ý với chúng. Nếu không có, lập trình viên thường phải đưa ra quyết định về yêu cầu trong quá trình lập trình. Yêu cầu rõ ràng giúp bạn tránh phải đoán xem người dùng muốn gì.
- Yêu cầu rõ ràng cũng giúp tránh xảy ra tranh cãi. Bạn quyết định về phạm vi của hệ thống trước khi bắt đầu lập trình. Nếu bạn xảy ra tranh cãi với một lập trình viên khác về điều gì chương trình cần phải làm, bạn có thể giải quyết bằng cách xem xét các yêu cầu viết thành văn bản.
- Việc chỉ định yêu cầu một cách đầy đủ là một yếu tố quan trọng đối với thành công của dự án, có lẽ thậm chí quan trọng hơn cả các kỹ thuật xây dựng hiệu quả.
- Yêu cầu ổn định là điều mục tiêu cao cả trong phát triển phần mềm. Với yêu cầu ổn định, một dự án có thể tiến từ kiến trúc, thiết kế, mã nguồn đến kiểm thử một cách có hệ thống, dự đoán và yên bình.
- Sự thay đổi bình thường có mức độ như thế nào? Các nghiên cứu tại IBM và các công ty khác đã phát hiện ra rằng dự án trung bình trải qua khoảng 25% thay đổi

trong yêu cầu trong quá trình phát triển, tương đương với 70 đến 85% công việc làm lại trên dự án thông thường.

- Để điều hướng các yêu cầu thay đổi hiệu quả trong quá trình xây dựng, hãy xem xét các chiến lược sau:
    - **Đánh giá chất lượng yêu cầu**: Sử dụng danh sách kiểm tra yêu cầu để đánh giá chất lượng của yêu cầu của bạn. Nếu chúng thiếu sót, tạm dừng việc mã hóa để sửa chúng trước khi tiến xa hơn.
    - **Làm nổi bật chi phí của những thay đổi**: Đảm bảo các bên liên quan hiểu rõ tác động của các thay đổi yêu cầu về mặt thời gian và chi phí. Đề xuất lịch trình và ước tính đã được điều chỉnh để đo đạc tính nghiêm trọng của các thay đổi đề xuất.
    - **Nhấn mạnh hiệu quả chi phí**: Nhấn mạnh rằng giải quyết các thay đổi trong giai đoạn yêu cầu thì rẻ hơn so với việc giải quyết chúng sau này. Sử dụng lập luận được trình bày trong chương này để nhấn mạnh điểm này.
    - **Thiết lập kiểm soát thay đổi**: Xem xét việc thiết lập một team quản lý (QA/PM) để xem xét và quản lý các thay đổi đề xuất, đảm bảo rằng các thay đổi thường xuyên có thể quản lý và được lập kế hoạch.
    - **Áp dụng phát triển linh hoạt**: Chọn các phương pháp phát triển mà có thể đáp ứng các thay đổi, như nguyên mẫu tiến hóa hoặc giao hàng tiến hóa. Những phương pháp này cho phép lặp lại, phản hồi và điều chỉnh.
    - **Tái đánh giá khả năng thực hiện dự án**: Nếu yêu cầu liên tục gặp vấn đề và các chiến lược trước không khả thi, xem xét hủy bỏ dự án. Xem xét ngưỡng mà bạn sẽ xem xét việc hủy bỏ và so sánh với tình hình hiện tại.
    - **Tập trung vào kế hoạch kinh doanh**: Luôn có mục tiêu kinh doanh của dự án trong tâm trí. Căn cứ yêu cầu với mục đích tổng thể của dự án. Xem xét giá trị kinh doanh gia tăng theo từng phần chứ không chỉ là việc thêm tính năng.
- Việc điều hướng các yêu cầu thay đổi đòi hỏi một phương pháp cân nhắc cân đối, bao gồm chất lượng, chi phí, tính thích ứng và mục tiêu kinh doanh.
- **Specific Functional Requirements**
    - Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?
    - Are all the outputs from the system specified, including their destination, accuracy, range of values, frequency, and format?
    - Are all output formats specified for Web pages, reports, and so on?
    - Are all the external hardware and software interfaces specified?
    - Are all the external communication interfaces specified, including handshaking, error-checking, and communication protocols?
    - Are all the tasks the user wants to perform specified?
    - Is the data used in each task and the data resulting from each task specified?
- **Specific Nonfunctional (Quality) Requirements**

  - Is the expected response time, from the user's point of view, specified for all necessary operations?
  - Are other timing considerations specified, such as processing time, datatransfer rate, and system throughput?
  - Is the level of security specified?
  - Is the reliability specified, including the consequences of software failure,
  - the vital information that needs to be protected from failure, and the strategy for error detection and recovery?
  - Are minimum machine memory and free disk space specified?
  - Is the maintainability of the system specified, including its ability to adapt to changes in specific functionality, changes in the operating environment, and changes in its interfaces with other software?
  - Is the definition of success included? Of failure?
- **Requirements Quality**
  - Are the requirements written in the user's language? Do the users think so?
  - Does each requirement avoid conflicts with other requirements?
  - Are acceptable tradeoffs between competing attributes specified—for example, between robustness and correctness?
  - Do the requirements avoid specifying the design?
  - Are the requirements at a fairly consistent level of detail? Should any requirement be specified in more detail? Should any requirement be specified in less detail?
  - Are the requirements clear enough to be turned over to an independent group for construction and still be understood? Do the developers think so?
  - Is each item relevant to the problem and its solution? Can each item be traced to its origin in the problem environment?
  - Is each requirement testable? Will it be possible for independent testing to determine whether each requirement has been satisfied?
  - Are all possible changes to the requirements specified, including the likelihood of each change?
- **Requirements Completeness**
  - Where information isn't available before development begins, are the areas of incompleteness specified?
  - Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
  - Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement and included just to appease your customer or your boss?

3.5. Architecture Prerequisite

- Thường thì, kiến trúc được mô tả trong một tài liệu duy nhất được gọi là 'mô tả kiến trúc' hoặc 'thiết kế cấp cao'. Một số người tạo ra sự phân biệt giữa kiến trúc

và thiết kế cấp cao - kiến trúc áp dụng các ràng buộc thiết kế trên toàn hệ thống, trong khi thiết kế cấp cao áp dụng các ràng buộc thiết kế ở cấp con hệ thống hoặc cấp đa lớp, nhưng không nhất thiết phải áp dụng trên toàn hệ thống.

- Tại sao cần có kiến trúc làm tiên quyết? Bởi vì chất lượng của kiến trúc xác định tính toàn vẹn khái niệm của hệ thống. Điều này lại xác định chất lượng cuối cùng của hệ thống. Một kiến trúc được suy nghĩ kỹ càng cung cấp cấu trúc cần thiết để duy trì tính toàn vẹn khái niệm của một hệ thống từ các cấp cao đến các cấp thấp. Nó cung cấp hướng dẫn cho các lập trình viên - ở mức chi tiết phù hợp với kỹ năng của các lập trình viên và công việc cụ thể. Nó chia thành các phần công việc để nhiều nhà phát triển hoặc nhiều nhóm phát triển có thể làm việc độc lập.
- Kiến trúc tốt làm cho quá trình xây dựng dễ dàng. Kiến trúc kém làm cho quá trình xây dựng gần như bất khả thi.

3.5.1. Thành phần kiến trúc điển hình

- Nhiều thành phần là chung cho các kiến trúc hệ thống tốt. Nếu bạn đang xây dựng toàn bộ hệ thống một mình, công việc về kiến trúc sẽ chồng chất với công việc thiết kế chi tiết hơn. Trong trường hợp như vậy, bạn ít nhất cần suy nghĩ về mỗi thành phần kiến trúc. Nếu bạn đang làm việc trên một hệ thống được thiết kế bởi người khác, bạn nên có khả năng tìm thấy các thành phần quan trọng mà không cần phải có một con chó săn mồi, một chiếc nón thám tử và một kính lúp. Trong cả hai trường hợp, đây là các thành phần kiến trúc cần xem xét.
- **Program Organization**:
    - Hệ thống kiến trúc hiệu quả cần có một cái nhìn tổng quan về hệ thống bằng những khía cạnh rộng. Thiếu cái nhìn tổng quan như vậy sẽ làm khó khăn việc hiểu cách các thành phần cá nhân kết hợp với nhau. Tương tự như việc giải một bức tranh ghép đơn giản, để hiểu một hệ thống phức tạp với nhiều hệ thống con yêu cầu một sự hiểu rõ rõ ràng về cách chúng tương tác.
    - Kiến trúc cần thể hiện các lựa chọn khác nhau cho cấu trúc cuối cùng và giải thích lý do chọn cách tổ chức. Quy trình này làm rõ vai trò của từng thành phần và giải thích cơ sở lựa chọn cấu trúc hệ thống. Quan điểm này quan trọng cho mục đích bảo trì, như được thể hiện trong thực hành thiết kế.
    - Xác định các khối xây dựng chính là một khía cạnh quan trọng của kiến trúc. Các khối này có thể là các lớp riêng lẻ hoặc các hệ thống con gồm nhiều lớp. Chúng cùng nhau xử lý các chức năng cấp cao như tương tác người dùng, hiển thị trang web, giải thích lệnh, bao gồm luật kinh doanh và truy cập dữ liệu, đảm bảo bao gồm mọi yêu cầu đã nêu trong yêu cầu. Nếu nhiều khối yêu cầu cùng một chức năng, các yêu cầu này nên bổ trợ lẫn nhau, không xung đột.
    - Mỗi khối xây dựng nên có trách nhiệm cụ thể, tập trung vào lĩnh vực được giao và hạn chế kiến thức về lĩnh vực của các khối xây dựng khác. Tiếp cận này giảm thiểu thông tin về thiết kế lan tràn và nâng cao khả năng bảo trì. Ngoài ra, quy tắc giao tiếp cho mỗi khối xây dựng cần được xác

định rõ, xác định các khối xây dựng khác mà khối xây dựng đó có thể tương tác trực tiếp hoặc gián tiếp, cũng như các khối không nên tương tác chung.

- **Major Classes**:
  - Kiến trúc cần xác định các lớp chính sẽ được sử dụng. Nó cần xác định trách nhiệm của từng lớp chính và cách lớp sẽ tương tác với các lớp khác. Nó cần bao gồm mô tả về cấu trúc lớp, các chuyển trạng thái, và bảo tồn đối tượng. Nếu hệ thống đủ lớn, nó cần mô tả cách các lớp được tổ chức thành các hệ thống con.
  - Kiến trúc cần mô tả các thiết kế lớp khác đã được xem xét và đưa ra lý do ưu tiên tổ chức đã được chọn. Kiến trúc không cần phải xác định từng lớp trong hệ thống. Hãy hướng tới quy tắc 80/20: xác định 20% lớp tạo nên 80% hành vi của hệ thống.
- **Data Design**:
  - Kiến trúc cần mô tả thiết kế các tệp và bảng chính sẽ được sử dụng. Nó cần mô tả các lựa chọn đã được xem xét và giải thích lý do của những lựa chọn đã được thực hiện. Nếu ứng dụng duy trì danh sách các ID khách hàng và các kiến trúc sư đã chọn biểu diễn danh sách ID bằng cách sử dụng danh sách truy cập tuần tự, tài liệu nên giải thích tại sao danh sách truy cập tuần tự tốt hơn so với danh sách truy cập ngẫu nhiên, ngăn xếp hoặc bảng băm. Trong quá trình xây dựng, thông tin như vậy giúp bạn hiểu thêm về tư duy của các kiến trúc sư. Trong quá trình bảo trì, cùng thông tin này là một nguồn hỗ trợ vô giá. Thiếu nó, bạn như đang xem một bộ phim nước ngoài mà không có phụ đề.
  - Dữ liệu thường nên được truy cập trực tiếp bởi một hệ thống hoặc lớp duy nhất, trừ khi thông qua các lớp hoặc gói truy cập cho phép truy cập vào dữ liệu một cách kiểm soát và trừu tượng.
  - Kiến trúc cần xác định high-level organization và nội dung ở bất kỳ cơ sở dữ liệu nào được sử dụng. Nó nên giải thích tại sao một cơ sở dữ liệu duy nhất ưu tiên hơn nhiều cơ sở dữ liệu khác(hoặc ngược lại), giải thích tại sao cơ sở dữ liệu ưu tiên hơn các tệp phẳng, xác định các tương tác có thể có với các chương trình khác truy cập cùng dữ liệu, giải thích các góc nhìn đã được tạo trên dữ liệu và còn nhiều nội dung khác.
- **Business Rules**:
  - Nếu kiến trúc phụ thuộc vào các quy tắc kinh doanh cụ thể, nó cần xác định và mô tả tác động mà những quy tắc này ảnh hưởng đến thiết kế của hệ thống. Ví dụ, giả sử hệ thống yêu cầu tuân thủ một quy tắc kinh doanh là thông tin khách hàng không được cũ hơn 30 giây. Trong trường hợp đó, cần mô tả tác động của quy tắc này đối với cách kiến trúc tiếp cận việc cập nhật và đồng bộ thông tin khách hàng.
- **User Interface Design**:
  - Giao diện người dùng thường được xác định tại thời điểm yêu cầu. Nếu không, nó nên được xác định trong kiến trúc phần mềm. Kiến trúc nên

xác định các yếu tố chính của định dạng trang web, giao diện đồ họa (GUI), giao diện dòng lệnh, và cả những yếu tố khác. Sự xây dựng kiến trúc cẩn thận cho giao diện người dùng là điểm khác biệt giữa một chương trình được yêu thích và một chương trình không bao giờ được sử dụng.
    - Kiến trúc nên được chia thành các mô-đun để có thể thay thế giao diện người dùng mới mà không ảnh hưởng đến business rules và output của chương trình. Ví dụ, kiến trúc nên tạo điều kiện tương đối dễ dàng để cắt bỏ một nhóm lớp giao diện tương tác và thay vào đó một nhóm lớp dòng lệnh.
- **Resource Management**:
    - Quản lý bộ nhớ cũng là một lĩnh vực quan trọng mà kiến trúc cần xử lý trong các lĩnh vực ứng dụng có hạn chế bộ nhớ như phát triển trình điều khiển và hệ thống nhúng. Kiến trúc nên ước tính tài nguyên được sử dụng cho các trường hợp thông thường và trường hợp extreme. Trong trường hợp đơn giản, các ước tính nên cho thấy tài nguyên cần thiết nằm trong khả năng của môi trường triển khai dự định. Trong trường hợp phức tạp hơn, ứng dụng có thể yêu cầu quản lý tài nguyên của chính nó một cách tích cực hơn.
- **Security**:
    - Kiến trúc nên mô tả phương pháp cho việc bảo mật cấp thiết kế và mã nguồn. Nếu một mô hình đe dọa trước đó chưa được xây dựng, thì nên xây dựng nó tại thời điểm kiến trúc. Hướng dẫn viết mã nên được phát triển với ý định về tác động đến bảo mật, bao gồm cách xử lý bộ đệm, quy tắc xử lý dữ liệu không đáng tin cậy (dữ liệu nhập vào từ người dùng, cookie, dữ liệu cấu hình và các giao diện bên ngoài khác), mã hóa, mức chi tiết chứa trong thông báo lỗi, bảo vệ dữ liệu bí mật trong bộ nhớ và các vấn đề khác.
- **Performance**:
    - Nếu hiệu suất là một vấn đề, các mục tiêu về hiệu suất nên được xác định trong yêu cầu. Các mục tiêu về hiệu suất có thể bao gồm việc sử dụng tài nguyên, trong trường hợp này các mục tiêu cũng nên xác định ưu tiên giữa các tài nguyên, bao gồm tốc độ so với bộ nhớ so với chi phí.
    - Kiến trúc nên cung cấp các ước tính và giải thích tại sao các kiến trúc sư tin rằng các mục tiêu có thể đạt được. Nếu các lĩnh vực cụ thể có nguy cơ không đáp ứng được mục tiêu của chúng, kiến trúc nên nêu ra điều đó. Nếu các lĩnh vực cụ thể yêu cầu sử dụng các thuật toán hoặc kiểu dữ liệu cụ thể để đáp ứng mục tiêu hiệu suất của chúng, kiến trúc nên nêu rõ điều đó. Kiến trúc cũng có thể bao gồm ngân sách về không gian và thời gian cho mỗi lớp hoặc đối tượng.
- **Scalability**:
    - Khả năng mở rộng (scalability) là khả năng của một hệ thống để tăng cường để đáp ứng các yêu cầu trong tương lai. Kiến trúc nên mô tả cách mà hệ thống sẽ đối phó với sự tăng trưởng về số người dùng, số máy chủ,

số nút mạng, số bản ghi cơ sở dữ liệu, kích thước bản ghi cơ sở dữ liệu, khối lượng giao dịch, và các yếu tố tương tự. Nếu không dự kiến hệ thống sẽ phát triển và khả năng mở rộng không phải là một vấn đề, kiến trúc nên làm rõ giả định đó.

- **Interoperability**:
  - Nếu dự kiến hệ thống sẽ chia sẻ dữ liệu hoặc tài nguyên với phần mềm hoặc phần cứng khác, kiến trúc nên mô tả cách mà điều đó sẽ được thực hiện.
- **Internationalization/Localization**:
  - Internationalization là hoạt động kỹ thuật để chuẩn bị một chương trình để hỗ trợ nhiều vùng địa lý khác nhau. Internationalization thường được biết đến với tên viết tắt "I18n" vì ký tự đầu và cuối cùng trong " Internationalization" là "I" và "N" và vì có 18 ký tự nằm ở giữa từ. Localization (được biết đến với tên viết tắt "L10n" với cùng lý do) là hoạt động dịch một chương trình để hỗ trợ một ngôn ngữ địa phương cụ thể.
  - Các vấn đề liên quan đến Internationalization xứng đáng được chú ý trong kiến trúc của một hệ thống tương tác. Hầu hết các hệ thống tương tác chứa hàng chục hoặc hàng trăm thông báo, màn hình trạng thái, thông báo trợ giúp, thông báo lỗi và những điều tương tự. Tài nguyên được sử dụng bởi các chuỗi văn bản nên được ước tính. Nếu chương trình được sử dụng trong mục đích thương mại, kiến trúc nên cho thấy rằng các vấn đề chuỗi văn bản và bộ ký tự phổ biến đã được xem xét, bao gồm việc sử dụng bộ ký tự nào (ASCII, DBCS, EBCDIC, MBCS, Unicode, ISO 8859, và cetera), các loại chuỗi văn bản được sử dụng (chuỗi C, chuỗi Visual Basic, và cetera), duy trì chuỗi văn bản mà không thay đổi mã nguồn, và dịch chuỗi văn bản sang các ngôn ngữ nước ngoài mà ảnh hưởng tối thiểu đến mã nguồn và giao diện người dùng. Kiến trúc có thể quyết định sử dụng chuỗi văn bản ngay trong mã nguồn nơi cần chúng, giữ các chuỗi trong một lớp và tham chiếu đến chúng thông qua giao diện lớp, hoặc lưu trữ các chuỗi trong một tệp tài nguyên. Kiến trúc nên giải thích tùy chọn nào đã được chọn và tại sao.
- **Input/Output**:
  - Kiến trúc nên xác định một hệ thống đọc trước, đọc sau hoặc đọc ngay khi cần. Và nó nên mô tả mức độ mà lỗi I/O được phát hiện: ở mức trường, bản ghi, luồng dữ liệu hoặc mức tệp.
- **Error Processing**: Xử lý lỗi đang trở thành một trong những vấn đề khó nhằn nhất trong khoa học máy tính hiện tại và không thể để xử lý lỗi một cách bừa bãi. Một số người đã ước tính rằng tới 90% mã của một chương trình được viết cho các trường hợp ngoại lệ, xử lý lỗi hoặc công việc quản lý, ngụ ý rằng chỉ có 10% được viết cho các trường hợp bình thường .Với nhiều mã dành cho việc xử lý lỗi như vậy, một chiến lược để xử lý chúng một cách nhất quán nên được nêu rõ trong kiến trúc. Xử lý lỗi thường được xem xét ở mức thực thi mã. Nhưng vì

nó có tác động toàn hệ thống, nên nó nên được xem xét ở mức kiến trúc. Dưới đây là một số câu hỏi cần xem xét:

- Xử lý lỗi có mục tiêu sửa lỗi hoặc chỉ là phát hiện lỗi? Nếu mục tiêu là sửa lỗi, chương trình có thể cố gắng khôi phục từ lỗi. Nếu chỉ là phát hiện lỗi, chương trình có thể tiếp tục xử lý như không có gì xảy ra, hoặc có thể thoát. Trong cả hai trường hợp, chương trình nên thông báo cho người dùng rằng nó đã phát hiện lỗi.
- Phát hiện lỗi là hoạt động tích cực hay thụ động? Hệ thống có thể dự đoán lỗi tích cực. Ví dụ, bằng cách kiểm tra đầu vào của người dùng để xác thực hoặc có thể thụ động phản ứng với lỗi chỉ khi không thể tránh chúng. Ví dụ, khi kết hợp đầu vào của người dùng tạo ra một tràn số. Nó có thể làm sạch hoặc dọn dẹp hậu quả. Một lần nữa, trong cả hai trường hợp, sự lựa chọn có tác động đến giao diện người dùng.
- Làm thế nào chương trình truyền tải lỗi? Sau khi phát hiện lỗi, chương trình có thể ngay lập tức loại bỏ dữ liệu gây ra lỗi, có thể xem lỗi như một lỗi và nhập vào trạng thái xử lý lỗi, hoặc có thể chờ đến khi tất cả quá trình xử lý hoàn thành và thông báo cho người dùng rằng đã phát hiện lỗi (ở đâu đó).
- Quy ước xử lý thông báo lỗi là gì? Nếu kiến trúc không chỉ định một chiến lược duy nhất và nhất quán, giao diện người dùng sẽ trở nên rối loạn như một bức tranh ghép bằng mì ống và đậu khô khác nhau ở các phần khác nhau của chương trình. Để tránh tình trạng như vậy, kiến trúc nên thiết lập các quy ước cho thông báo lỗi.
- Làm thế nào các ngoại lệ sẽ được xử lý? Kiến trúc nên xem xét khi nào mã có thể gây ra ngoại lệ, nơi chúng sẽ được bắt, cách chúng sẽ được ghi nhật ký, cách chúng sẽ được tài liệu hóa, và cách khác.
- Trong chương trình, ở mức nào lỗi được xử lý? Bạn có thể xử lý chúng tại điểm phát hiện, chuyển giao chúng cho một lớp xử lý lỗi, hoặc chuyển gửi chúng lên chuỗi gọi.
- Mức trách nhiệm của từng lớp trong việc xác thực dữ liệu đầu vào của nó là gì? Mỗi lớp có trách nhiệm xác thực dữ liệu của riêng nó, hoặc có một nhóm lớp chịu trách nhiệm xác thực dữ liệu của hệ thống? Các lớp ở bất kỳ mức nào có thể cho rằng dữ liệu mà họ nhận được là sạch không?
- Bạn có muốn sử dụng cơ chế xử lý ngoại lệ tích hợp trong môi trường của bạn hay xây dựng riêng của bạn? Sự thật là môi trường có một phương pháp xử lý lỗi cụ thể không đồng nghĩa rằng đó là phương pháp tốt nhất cho yêu cầu của bạn.
- **Fault Tolerance**: Kiến trúc cũng nên chỉ ra loại khả năng chịu lỗi dự kiến. Khả năng chịu lỗi là một tập hợp các kỹ thuật làm tăng độ tin cậy của hệ thống bằng cách phát hiện lỗi, khôi phục lỗi nếu có thể và chứa các tác động xấu của chúng nếu không.

- **Architectural Feasibility**:

  - Các nhà thiết kế có thể lo ngại về khả năng của hệ thống đáp ứng các mục tiêu về hiệu suất, hoạt động trong giới hạn tài nguyên hoặc được hỗ trợ đầy đủ bởi môi trường triển khai. Kiến trúc nên chứng minh rằng hệ thống có khả năng kỹ thuật. Nếu không khả thi trong bất kỳ lĩnh vực nào có thể làm cho dự án không thể thực hiện được, kiến trúc nên chỉ ra cách những vấn đề đó đã được điều tra qua các nguyên mẫu chứng minh khái niệm, nghiên cứu hoặc các phương tiện khác. Những rủi ro này nên được giải quyết trước khi bắt đầu xây dựng tỷ lệ lớn.
- **Overengineering**:
  - Khả năng chắc chắn (robustness) là khả năng của một hệ thống tiếp tục hoạt động sau khi phát hiện lỗi. Thường thì kiến trúc chỉ định một hệ thống có khả năng chắc chắn hơn so với yêu cầu. Một nguyên nhân là hệ thống được tạo thành từ nhiều phần tử mà mỗi phần tử đều có khả năng chắc chắn tối thiểu, có thể dẫn đến mức độ chắc chắn không đủ cao như yêu cầu tổng thể. Kiến trúc nên cho thấy rõ ràng liệu người lập trình có nên chọn phương pháp thiết kế overengineering hay chọn phương pháp đơn giản nhất mà vẫn hoạt động được.
  - Việc chỉ định một phương pháp thiết kế overengineering đặc biệt quan trọng, vì nhiều lập trình viên thường tự động thiết kế overengineering cho các lớp của họ, vì lòng tự tôn nghề nghiệp. Bằng cách thiết lập kỳ vọng một cách rõ ràng trong kiến trúc, bạn có thể tránh hiện tượng một số lớp có khả năng chắc chắn đặc biệt cao và một số khác chỉ đáp ứng đủ yêu cầu tối thiểu.
- **Buy-vs.-Build Decisions**:
  - Nếu kiến trúc không sử dụng các thành phần sẵn có, nó nên giải thích cách mà nó kỳ vọng các thành phần do chính tay xây dựng sẽ vượt qua các thư viện và thành phần sẵn có.
- **Reuse Decisions**:
  - Nếu kế hoạch đề xuất sử dụng phần mềm đã tồn tại, các trường hợp kiểm thử, định dạng dữ liệu hoặc tài liệu khác, kiến trúc nên giải thích cách phần mềm được tái sử dụng sẽ được tuân thủ các mục tiêu kiến trúc khác.
- **Change Strategy**:
  - Một trong những thách thức chính mà một kiến trúc sư phần mềm phải đối mặt là làm cho kiến trúc linh hoạt đủ để thích nghi với các thay đổi có thể xảy ra.
  - Kiến trúc nên rõ ràng mô tả một chiến lược để xử lý các thay đổi. Kiến trúc nên cho thấy rằng các cải tiến có thể có đã được xem xét và rằng các cải tiến có khả năng cao cũng là những thứ dễ thực hiện nhất. Nếu các thay đổi có thể xảy ra trong định dạng đầu vào hoặc đầu ra, phong cách tương tác với người dùng, hoặc yêu cầu xử lý, kiến trúc nên cho thấy rằng tất cả các thay đổi đã được dự đoán và hiệu ứng của bất kỳ thay đổi đơn lẻ nào sẽ bị giới hạn trong một số lượng nhỏ các lớp. Kế hoạch của kiến trúc cho các thay đổi có thể đơn giản như đặt số phiên bản trong các tập

tin dữ liệu, dành các trường cho sử dụng tương lai, hoặc thiết kế tập tin để bạn có thể thêm mới các bảng. Nếu có sử dụng trình tạo mã, kiến trúc nên cho thấy rằng các thay đổi dự đoán nằm trong khả năng của trình tạo mã.

- Kiến trúc nên chỉ ra các chiến lược được sử dụng để trì hoãn cam kết. Ví dụ, kiến trúc có thể xác định rằng nên sử dụng kỹ thuật dựa vào bảng thay vì mã cứng nếu kiểm tra. Nó có thể xác định rằng dữ liệu cho bảng nên được giữ trong một tập tin bên ngoài thay vì được mã hóa bên trong chương trình, cho phép thay đổi trong chương trình mà không cần biên dịch lại.

- **General Architectural Quality**:
  - Một đặc điểm của một mô tả kiến trúc tốt là việc thảo luận về các lớp trong hệ thống, thông tin được ẩn trong mỗi lớp và lý do cho việc bao gồm và loại trừ tất cả các phương án thiết kế có thể.
  - Kiến trúc nên là một khái niệm tổng thể hoàn chỉnh với ít thêm vào tự phát. Một kiến trúc tốt nên phù hợp với vấn đề. Khi bạn nhìn vào kiến trúc, bạn nên hài lòng bởi cách giải pháp tự nhiên và dễ dàng mà nó đem lại. Nó không nên trông như vấn đề và kiến trúc đã được ghép cùng nhau bằng băng dính.
  - Mỗi thay đổi nên phù hợp một cách sạch sẽ với khái niệm tổng thể.
  - Mục tiêu của kiến trúc nên được nêu rõ. Một thiết kế cho hệ thống với mục tiêu chính là khả năng sửa đổi sẽ khác với một mục tiêu là hiệu suất không bị bất kỳ khoản đánh đổi nào, ngay cả khi cả hai hệ thống có cùng chức năng.
  - Kiến trúc nên mô tả động cơ cho tất cả các quyết định quan trọng.
  - Kiến trúc phần mềm tốt đa phần là độc lập với máy và ngôn ngữ.

3.5.2. List of issues that a good architecture should address.

- **Specific Architectural Topics**
  - Is the overall organization of the program clear, including a good architectural overview and justification?
  - Are major building blocks well defined, including their areas of responsibility and their interfaces to other building blocks?
  - Are all the functions listed in the requirements covered sensibly, by neither
  - too many nor too few building blocks?
  - Are the most critical classes described and justified?
  - Is the data design described and justified?
  - Is the database organization and content specified?
  - Are all key business rules identified and their impact on the system described?
  - Is a strategy for the user interface design described?
  - Is the user interface modularized so that changes in it won't affect the rest of the program?

- Is a strategy for handling I/O described and justified?
- Are resource-use estimates and a strategy for resource management
- described and justified for scarce resources like threads, database connections, handles, network bandwidth, and so on?
- Are the architecture's security requirements described?
- Does the architecture set space and speed budgets for each class, subsystem, or functionality area?
- Does the architecture describe how scalability will be achieved?
- Does the architecture address interoperability?
- Is a strategy for internationalization/localization described?
- Is a coherent error-handling strategy provided?
- Is the approach to fault tolerance defined (if any is needed)?
- Has technical feasibility of all parts of the system been established?
- Is an approach to overengineering specified?
- Are necessary buy-vs.-build decisions included?
- Does the architecture describe how reused code will be made to conform to other architectural objectives?
- Is the architecture designed to accommodate likely changes?

- **General Architectural Quality**
  - Does the architecture account for all the requirements?
  - Is any part overarchitected or underarchitected? Are expectations in this area set out explicitly?
  - Does the whole architecture hang together conceptually?
  - Is the top-level design independent of the machine and language that will be used to implement it?
  - Are the motivations for all major decisions provided?
  - Are you, as a programmer who will implement the system, comfortable
  - with the architecture?

3.6. Amount of Time to Spend on Upstream Prerequisites

- Tập trung vào việc quản lý thời gian và kế hoạch cho việc định nghĩa vấn đề, yêu cầu và kiến trúc phần mềm trong các dự án phát triển phần mềm. Nội dung chính bao gồm những phần như
- Phân chia thời gian cho việc định nghĩa vấn đề, yêu cầu và kiến trúc phần mềm:
  - Thời gian dành cho định nghĩa vấn đề, yêu cầu và kiến trúc phần mềm phụ thuộc vào nhu cầu của dự án.
  - Một dự án hiệu quả thường dành khoảng 10-20% công sức và 20-30% thời gian lên kế hoạch ban đầu cho yêu cầu, kiến trúc và lập kế hoạch.
- Xử lý yêu cầu không ổn định:
  - Nếu yêu cầu không ổn định và dự án lớn, cần làm việc với một chuyên gia phân tích yêu cầu để giải quyết vấn đề yêu cầu phát hiện sớm trong giai đoạn xây dựng.
  - Nếu yêu cầu không ổn định và dự án nhỏ, bạn cần giải quyết vấn đề yêu cầu một cách tự quản.

- Xác định thời gian cho việc thiết kế kiến trúc phần mềm:
    - Phân bổ thời gian cho kiến trúc phần mềm tương tự như việc phát triển yêu cầu.
    - Đối với các loại phần mềm mới, cần phải dành thêm thời gian để xây dựng kiến trúc trong lĩnh vực mới.
    - Đảm bảo thời gian cần thiết để tạo kiến trúc tốt không làm giảm bớt thời gian cần cho các công việc khác.
- Kế hoạch xử lý yêu cầu và kiến trúc phần mềm như các dự án riêng biệt nếu cần.
- So sánh với việc xây dựng công trình vật liệu:
    - Giải thích rằng việc đặt ra yêu cầu hoặc xây dựng mà không có kế hoạch trước đó là không hợp lý, giống như việc xây dựng một công trình mà không có bản thiết kế trước.
- Yêu cầu giải thích lý do lập kế hoạch cho phát triển yêu cầu và kiến trúc phần mềm riêng biệt cho khách hàng không hiểu rõ về phát triển phần mềm.