**Code Complete 2**

**Part 1: Laying the Foundation**

**Chapter 1: Welcome to Software Construction**

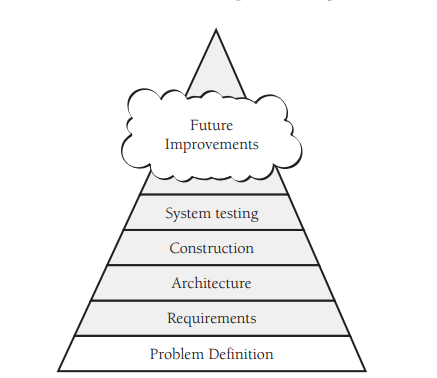
* Quá trình xây dựng (construction): lên kế hoạch, thiết kế, kiểm tra
* Xây dung phần mềm: coding và debugging, detailed design, construction planning, itegration, developer testing (unit testing, integration testing)
* Xây dựng phần mềm còn đc gọi là: coding, programming.
* Vai trò của xây dựng phần mềm:
  + Là 1 phần lớn của quá trình phát triển phần mềm
  + Là giai đoạn trung tâm: sau giai đoạn kiến trúc, trước giai đoạn kiểm tra (system testing)
  + Tăng hiệu suất làm việc của từng PG nếu có kĩ thuật tốt
  + Là giai đoạn chắc chắn đc làm, nếu cải thiện giai đoạn này thì sẽ có áp dụng cho mọi project

**Chapter 2: Metaphors for a Richer Understanding of Software Development**

* Sử dụng phép ẩn dụ để hiểu rõ hơn về phần mềm
* Khác biệt giữa algorithm và heuristic: algorithms chỉ ra giải pháp trực tiếp, heuristic chỉ dẫn cách tìm ra giải pháp
* Incremental development: đầu tiên tạo ra hệ thống đơn giản nhất có thể, ko cần dữ liệu thật, kết quả thật, nhưng phải là 1 khung xương vững chắc để phát triển hệ thống.
* Dựa trên khung xương này, ta sẽ thêm những code từng chút một cho đến khi đầy đủ
* Khi làm dự án nhỏ, nếu có sai sót thì có thể sửa nhanh chóng hoặc làm lại từ đầu, nhưng nếu là dự án lớn, thời gian sẽ rất lâu hơn và hậu quả có thể lớn hơn
* Cần cấu trúc, thiết kế tốt chương trình ngay từ đầu
* Vậy làm thế nào để có thiết kế tốt ngay từ đầu? Dự đoán? Cấu trúc dữ liệu? Giải thuật?
* Không thể tự mình xây mọi thứ mà phải kế thừa, dùng những cái đc xây sẵn (thư viện), nhưng cũng có thể tự mình viết lại những chức năng nếu nó là 1 trong các ưu tiên (về performance)
* Lên kế hoach cẩn thận ko có nghĩa là quá chi tiết. 1 kế hoạch tốt có thể cho phép thay đổi,phát triển dễ dàng
* Kĩ thuật không phải là những qui tắc, mà là công cụ.
* Cần biết trong trường hợp nào cần dùng công cụ nào
* 1 số thành công hay thất bại đc quyết định trc khi giai đoạn construction bắt đầu.

**Chapter 3: Measure Twice, Cut Once: Upstream Prerequisites.**

* Measure twice, cut once: làm 1 lần construction thui.
* Để tạo ra phần mềm tốt, cần chú trọng các giai đoạn đầu, giữa và cuối.
* Chú trọng giai đoạn cuối nghĩa là kiểm tra phần mềm. Điều này chỉ đóng vai trò nhỏ trong chất lượng sản phẩm.
* Trong giai đoạn construction, ta cần phải nhận biết đc đang ở trong tình hình ntn (kế thừa từ giai đoạn trc đó)
* Cần xác định mình đã sẵn sàng vào giai đoạn construction chưa.
* Dự án sẽ chạy tốt nếu những sự chuẩn bị thích hợp đc thực hiện trc khi vào xây dựng.
* Mục tiêu của sự chuẩn bị: giảm rủi ro.
* Sự chuẩn bị: cải thiện yêu cầu, kế hoạch.
* Nếu 1 việc ko đc làm tốt ngay từ đầu, có làm nhiều việc # cũng ko có ích?
* Hội chứng WISCA: chỉ muốn nhảy vào code mà ko có sự chuẩn bị tốt.
* Định nghĩa vấn đề, yêu cầu, kiến trúc
* Trc khi làm 1 hệ thống, cần biết hệ thống đó sẽ làm cái gì và sẽ làm như thế nào.
* Từ góc độ quản lý, lên kế hoạch nghĩa là quyết định số thời gian, số người, số máy tính, …
* Góc độ kĩ thuật, lên kế hoạch nghĩa là hiểu cái cần làm để ko làm sai.
* Cần dự đoán và sửa các lỗi ngay khi nó xuất hiện, càng để lâu thì hậu quả càng lớn. Do đó, nếu có lỗi trong các yêu cầu, mà do xác định yêu cầu là bước đầu, nên nếu để lỗi đó tồn tại lâu thì sẽ ảnh hưởng lớn.
* **Điều kiện tiền đề: định nghĩa vấn đề (problem definition)**
* Trong định nghĩa vấn đề ko có đề cập đến giải pháp



* Phải diễn đạt = ngôn ngữ thông thường, từ góc độ người dùng
* **Điều kiện tiền đề: các yêu cầu của hệ thống**
* Giúp PG khỏi phải đoán user muốn gì
* Tránh tranh cãi
* Là điều quan trọng để dự án thành công, quan trọng hơn cả những kĩ thuật xây dựng phần mềm
* Yêu cầu ổn định là cực kỳ quan trọng
* Các câu hỏi về yêu cầu:
  + **Những yêu cầu chức năng cụ thể**
  + Tất cả những đầu vào hệ thống đã được xác định chưa? Bao gồm nguồn, tính xác thực, phạm vi, tần số vào.
  + Tất cả những đầu ra của hệ thống đã được xác định chưa? Bao gồm đích, tính chính xác, phạm vi, tần số và định dạng.
  + Phần cứng, phần mềm đã đc chỉ rõ?
  + Những cách thức giao tiếp đã đc chỉ rõ? Gồm hand-shaking, error-checking, communication protocol
  + Tất cả những công việc mà người dùng muốn thực hiện đã đc chỉ rõ?
  + Đầu vào và đầu ra của từng công việc đã đc chỉ rõ?
  + **Những yêu cầu phi chức năng**
  + Từ góc độ người dùng, thời gian đáp ứng mong muốn cho các hoạt động đã đc chỉ rõ?
  + Những xem xét thời gian khác đã đc chỉ ra? Như thời gian xử lý, tốc độ truyền dữ liệu, lưu lương hệ thống
  + Mức độ bảo mật đã đc chỉ rõ?
  + Độ tin cậy đã đc chỉ rõ? Như những hậu quả khi phần mềm bị sai (?), những thông tin quan trọng, phương pháp cho phát hiện lỗi và phục hồi.
  + Dung lượng bộ nhớ, ổ đĩa cần thiết đã đc chỉ rõ?
  + Tính bảo trì của hệ thống đã đc chỉ rõ? Gồm khả năng đáp ứng thay đổi của các chức năng, thay đổi os, thay đổi giao tiếp với các phần mềm khác.
  + Định nghĩa thành công, thất bại?
  + **Chất lượng các yêu cầu**
  + Các yêu cầu có đc viết = ngôn ngữ của người dùng? Người dùng có thật sự nghĩ như vậy?
  + Các yêu cầu có va chạm với nhau ko?
  + Sự cân bằng (nếu có va chạm) có đc chỉ rõ?
  + Trong yêu cầu có chỉ ra cách thiết kế ko?
  + Các yêu cầu đã đc mô tả ở mức độ cần thiết ko? Có cần chi tiết hơn hoặc ít chi tiết hơn?
  + Các yêu cầu có dễ hiểu ko?
  + Các yêu cầu có kiểm tra đc ko? Có kiểm tra độc lập đc ko?
  + Tất cả những thay đổi có thể có đã đc chỉ rõ? Khả năng xảy ra sự thay đổi đó?
  + **Sự đầy đủ của yêu cầu**
  + Khi bắt đầu làm mà các yêu cầu chưa đầy đủ, những vùng (chức năng, phạm vi) chưa có đã đc chỉ rõ?
  + Tất cả những yêu cầu có làm mình hài lòng?
* Các yêu cầu cua kh có thể thay đổi trong quá trình code
  + Khi yêu cầu thay đổi, cần ngừng code và làm lại yêu cầu cho đúng.
  + Cho người khác biết tài nguyên phải tốn khi thay đổi yêu cầu
  + Sử dụng những phương pháp thích nghi với sự thay đổi
* **Điều kiện tiền đề: kiến trúc hệ thống**
* Kiến trúc phần mềm là 1 phần cao cấp của thiết kế phần mềm
* Kiến trúc xác định sự nhất quán của hệ thống
* Kiến trúc phân chia công việc, các nhóm có thể làm việc độc lập.
* **Những thành phần kiến trúc**
  + **Tổ chức chương trình**
  + Mô tả tổng quan về hệ thống
  + Tìm kiểm những cách thay thế khác, và đưa ra lí do tại sao lại chọn cách cuối cùng
  + Định nghĩa những khối quan trọng của chương trình, mỗi khối có thể là 1 class hoặc 1 hệ thống nhỏ khác chứa nhiều class
  + Chức năng mỗi khối cần rõ rang. Mỗi khối biết chức năng của các khối # ít nhất có thể
  + Cách giao tiếp giữa các khối cần đc chỉ rõ, 1 khối có thể sử dụng những khối nào trực tiếp, gián tiếp, hoặc ko đc sử dụng.
  + **Các class chính**
  + **Thiết kế dữ liệu**
  + **Qui tắc kinh doanh**
  + **Thiết kế giao diện người dùng**
  + **Quản lý tài nguyên**
  + Ước lượng những trường hợp cực nhỏ hoặc cức lớn
  + **Bảo mật**
  + Mô tả phương phapr bảo mật ở cấp độ thiết kế và cấp độ lập trình.
  + Quản lý buffer, xử lý những dữ liệu ko đáng tin (đầu vào từ user, cookie, giá trị config, …), mã hóa, độ chi tiết của thông báo lỗi, dữ liệu bí mật, …
  + **Hiệu suất (performance)**
  + Nếu hiệu suất cần đc quan tâm thì nên đc chỉ rõ trong khi tạo các yêu cầu
  + Gồm việc sử dụng tài nguyên, tốc độ, bộ nhớ, chi phí
  + Kiến trúc cần đưa ra những ước lượng và giải thích tại tại sao có thể đạt đc mục tiêu performance đó
  + Kiến trúc cần chỉ ra những chỗ khó đạt đc hiệu suất cần thiết, những chỗ cần những giải thuật, cấu trúc đặc biệt để đạt đc hiệu suất
  + **Khả năng mở rộng (scalability)**
  + Xử lý ntn nếu hệ thống tăng lên về số lượng người dùng, số lượng server, node mạng, db, …
  + Nếu khả năng mở rộng ko phải là 1 yêu cầu thì cần chỉ rõ.
  + **Sự tương tác**
  + **Quốc tế hóa/ Địa phương hóa**
  + Hỗ trợ nhiều nơi trên thế giới
  + **Đầu vào/ Đầu ra**
  + Xác định đầu vào, đầu ra
  + Xác đinh mức độ phát hiện lỗi, ở record, file, stream, ..
  + **Xử lý lỗi**
  + Các câu hỏi:
    - Xử lý lỗi ở mức độ khắc phục hay phát hiện? Mức độ phát hiện có nghĩa là ko sửa mà chỉ bỏ qua hoặc ngừng lại, có thể thông báo cho người dùng, mức độ khắc phục là hệ thống sẽ tự động khắc phục lỗi đó
    - Phát hiện lỗi là chủ động hay bị động? Chủ động nghĩa là có thể kiểm tra trc dữ liệu, bị động là lỗi xảy ra mới biết
    - Cách chương trình lan truyền lỗi (propagate errors)
    - Qui ước cho xử lý lỗi là gì? Kiến trúc cần qui định 1 cách nhất quán
    - Exception đc xử lý ntn? Khi nào thảy exception, ở đâu? Ghi log ntn? …
    - Lỗi đc xử lý ở cấp độ nào? Ngay khi phát hiện, hoặc đưa nó cho class xử lý, hoặc truyền nó cho 1 chuỗi #?
    - Trách nhiệm của từng class trong việc kiểm tra dữ liệu. Mỗi class có cần kiểm tra dữ liệu của chính nó? Những class nào kiểm tra dữ liệu hệ thống? Có class nào ko cần kiểm tra dữ liệu ko?
    - Sử dụng những cách xử lý lỗi có sẵn của môi trường hay tự xây dựng lại.
  + **Khả năng chịu lỗi (fault tolerance)**
  + Kiến trúc cần chỉ ra hệ thống chịu đựng lỗi ntn?
  + Chịu đựng lỗi bao gồm các kỹ thuật phát hiện, phục hồi (nếu có thể), chịu đựng ảnh hưởng xấu
  + Hệ thống có thể xử lý lỗi theo các cách sau:
    - Hệ thống có thể quay lại và thử lại lần nữa nếu phát hiện lỗi. Điểm mà hệ thống quay lại là điểm mà ko có lỗi j xảy ra
    - Hệ thống có thể code phụ trợ nếu nó phát hiện lỗi trong code chính
    - Dùng giải thuật bỏ phiếu. Ví dụ, hệ thống có 3 class cùng làm 1 chức năng, ta có thể so sánh kết quả của 3 class này để lựa chọn
    - Hệ thống có thể thay giá trị lỗi = giá trị giả mạo # để những phần còn lại của hệ thống vẫn hoạt động tốt
  + **Tính khả thi**
  + Về mục tiêu hiệu suất, tài nguyên có hạn, môi trương thực hiện
  + Kiến trúc phải chỉ ra hệ thống có thể thực hiện đc
  + **Overengineering**
  + **Lựa chọn Buy-vs-Build**
  + **Quyết định dùng lại**
  + **Chiến lược thay đổi**
  + **Chất lượng kiến trúc**
  + 1 kiến trúc tốt thể hiện qua các lí do tại sao chọn lựa cái này mà ko phải cái kia, chức năng các class trong hệ thống, những thông tin của class cần đc giấu, …
  + Kiến trúc tốt thì phải tự nhiên, và dễ dàng
  + Mỗi sự thay đổi trong tương lai phải gắn với những khái niêm, phạm vi của kiến trúc ?
  + Mục tiêu của kiến trúc phải đc khẳng định rõ.
  + Kiến trúc phải chỉ ra động lực cho những quyết định. Phải có lí do cho những quyết định
  + Kiến trúc tốt thì độc lập với ngôn ngữ lập trình, môi trường.
  + Khi thiết kế kiến trúc, chỉ làm những việc cần thiết, chứ ko làm dư, làm những công việc của những giai đoạn #
  + Kiến trúc cần chỉ ra những khu vực (những thứ, chức năng) rủi ro. Lí do. Cách giảm thiểu rủi ro đó
  + Cần nhìn kiến trúc dưới nhiều góc độ khác nhau
* Danh sách câu hỏi:
  + **Kiến trúc:**
  + Cách tổ chức đã rõ ràng chưa?
  + Các khối chức năng chính đã được xây dựng tốt chưa? Bao gồm nhiệm vụ, và giao diện với các khối khác
  + Kiến trúc đã thực hiện hết các yêu cầu chưa? Liệu có quá nhiều khối hay quá ít khối cùng thực hiện 1 yêu cầu?
  + Đã mô tả và đánh giá những class quan trọng
  + Đã mô tả và đánh giá việc thiết kế dữ liệu
  + Đã chỉ ra cách thiết kế db
  + Đã quan tâm đến những nguyên tắc kinh doanh chưa?
  + Đã thiết kế giao diện người dùng chưa?
  + Để những thay đổi ở giao diện người dùng ko ảnh hưởng đến phần còn lại của chương trình, ta đã chuẩn hóa nó chưa?
  + Cách xử lý I/O đã đc mô tả và đánh giá chưa?
  + Đã ước lượng tài nguyên chưa? Đã có cách quản lý tài nguyên chưa?
  + Bảo mật ntn?
  + Tài nguyên cho mỗi class, mỗi khối đã đc chỉ rõ?
  + Kiến trúc đã chỉ ra cách mở rộng dự án chưa?
  + Interoperability
  + Quốc tế hóa/ Địa phương hóa
  + Cách thức xử lý lỗi đã đc chỉ ra và đánh giá chưa?
  + Đã có phương pháp chịu đựng lỗi chưa?
  + Hệ thống có khả thi hay ko?
  + Overengineering
  + Có những chức năng cần tận dụng từ cái có sẵn ko? hay là tự làm hết
  + Kiến trúc đã mô tả cách tạo ra code có thể sử dụng lại chưa?
  + Kiến trúc có khả năng thích nghi với các thay đổi tương lai ko?
  + **Chất lượng kiến trúc**
  + Kiến trúc đã xét đến tất cả yêu cầu chưa?
  + Có phần nào dư hay ko?
  + Kiến trúc có độc lập với ngôn ngữ, với môi trường ko?
  + Động lực để đưa ra quyết định là gì?
  + Mình có hài long với kiến trúc này chưa?
* **Amount of time to spend on Upstream Prerequisites**
* Danh sách câu hỏi:
* Đã xác định dự án mình thuộc loại nào?
* Tất cả yêu cầu đã đc mô tả tốt chưa, đã đủ ổn định chưa
* Kiến trúc đã tốt?
* Những rủi ro đã đc mô tả?

**Chapter 4: Key Construction Decisions**

**Choice of programming language**

* Những từ ngữ trong ngôn ngữ lập trình dùng để diễn đạt suy nghĩ của người lập trình

**Programming Conventions**

* Tên biến, tên class, tên routines, định dạng, chú thích, …

**Your Location on the Technology Wave**

* Phụ thuộc vào công nghệ mà mình lựa chọn đang ở giai đoạn nào, đã phát triển hay chưa, mà công việc mình sẽ khác nhau. Nếu mình đang ở giai đoạn đầu của công nghệ thì sẽ khó lập trình các chức năng hơn.
* Program in language: lập trình dựa vào ngôn ngữ
* Program into language: Suy nghĩ trước, sau đó dùng ngôn ngữ lập trình thực hiện cái mình muốn
* Nếu ngôn ngữ thiếu những đặc tính mà mình muốn, hãy tự tạo ra nó
* Tự tạo ra những qui tắc, qui ước, chuẩn khi lập trình

**Selection of Major Construction Practices**

* Danh sách câu hỏi về cách lập trình
* Đã định nghĩa các qui ước về tên, chú thích, cách trình bày?
* Đã định nghĩa những cách thức như xử lý lỗi, quan tâm đến bảo mật ntn, class interface, performance, …
* Đã biết công nghệ mình sử dụng đang ở giai đoạn nào? Cần điều chỉnh cách thức cho phù hợp
* Đã xác định đc cách sử dụng ngôn ngữ để thực hiện qui tắc lập trình?
* Đã định nghĩa qui trình nhất quán chưa?
* Có cần vừa lập trình vừa test hay ko?
* Có cần viết testcase trc khi code?
* Có cần viết unit test?
* Có cần dùng debugger để coi quá trình chạy chương trình?
* Integration test?
* Có cần xem xét code của người trong nhóm?
* Đã cọn công cụ quản lý phiên bản?
* Đã chọn ngôn ngữ, phiên bản của ngôn ngữ?
* Có dùng fw ko? Nếu có thì chọn fw nào?
* Sử dụng những đặc tính ko chuẩn của ngôn ngữ?
* Những công cụ khác: trình soạn thảo, refactor, debugger, …
* Cần lập trình sử dụng ngôn ngữ, chứ ko lập trình theo ngôn ngữ

**Part 2: Creating High-Quality Code**

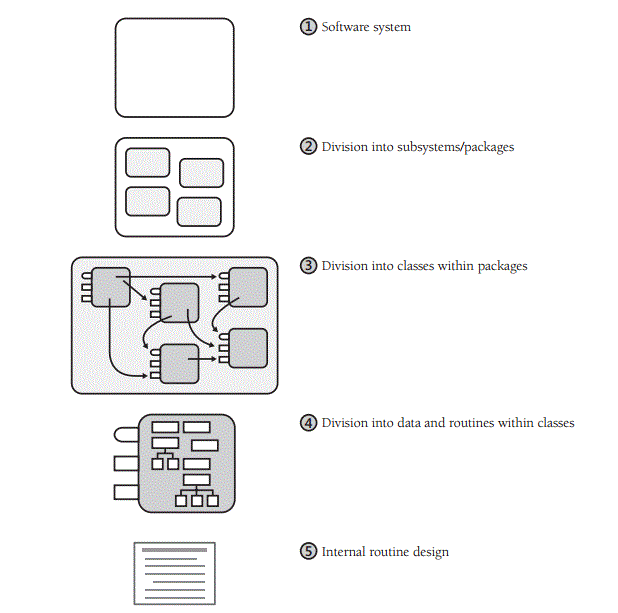
**Chapter 5: Design in Construction**

**Design Challenges**

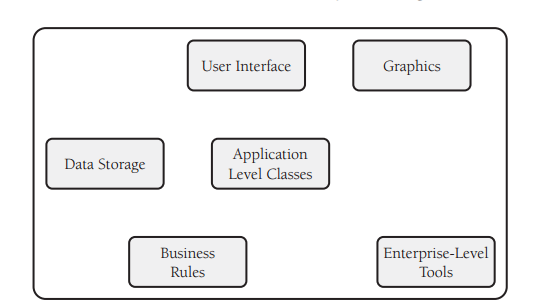
* Thiết kế phần mềm là khái niệm, là sự chế tạo 1 mô hình, chuyển yêu cầu kĩ thuật thành phần mềm
* 1 thiết kế cấp cao tốt (kiến trúc) sẽ cung cấp 1 cấu trúc có thể chứa những thiết kế nhỏ hơn
* **Design is a wicked problem**
* Không có thiết kế nào ko bị lỗi, mình phải thực hiện cái thiết kế đó, xem xét những lỗi xảy ra, và làm lại thiết kế khác tốt hơn ???
* **Design is a sloppy process (Even if it produces tidy results)**
* Thiết kế cuối cùng sẽ hoàn thiện, có cấu trúc tốt, nhưng quá trình tạo ra nó rất là lằng nhằng.
* Phạm lỗi trong quá trình thiết kế, sau đó sửa lại sẽ tốt hơn phạm lỗi trong quá trình code
* 1 giải pháp tốt hơn thường khác so với giải pháp cũ chỉ ở 1 vài chỗ
* Khó để có thể biết 1 thiết kế đã đủ tốt hay chưa nên vấn đề là mình có còn thời gian cho thiết kế hay chưa
* **Design is about Tradeoffs and Priotities**
* **Design involves restrictions**
* Thiết kế là tạo ra các khả năng và hạn chế các khả năng
* Sự giới hạn về tài nguyên (thời gian, bộ nhớ, băng thông, …) sẽ thúc đẩy việc tạo ra những giải pháp tốt hơn.
* **Design is Nondeterministic**
* Có rất nhiều cách để thiết kế 1 chương trình
* **Design is a Heuristic Process**
* **Design is Emergent**

**Key Design Concepts**

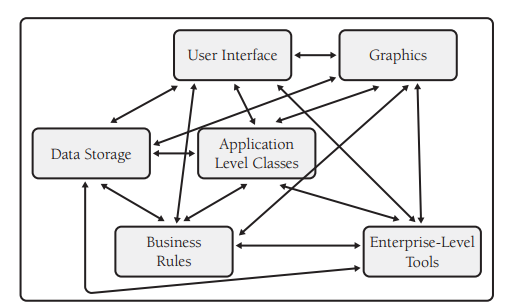
* Muốn thiết kế tốt cần hiểu rõ các khái niệm quan trọng.
* **Software’ Primary Technical Imperative: Managing Complexity**
  + **Accidental and Essential Difficulties**
  + Quá trình phát triển phần mềm gặp khó khăn bởi vì 2 vấn đề: essential và accidental
  + Essential của 1 vật là đặc tính của vật mà nếu nó ko có thì nó không phải là nó @@@
  + Accidental: là những đặc tính lựa chọn (optional), mỗi vật sẽ khác nhau
  + **Importance of Managing Complexity**
  + Nếu 1 dự án thất bại vì yếu tố kỹ thuật (ngoài yếu tố yêu cầu, kế hoạch, quản lý) thì đó là vì không kiểm soát đc sự phức tạp
  + Nên tổ chức chương trình mà ở 1 thời điểm, ta chỉ cần tập trung vào 1 phần
  + Do đó mục tiêu là giảm lượng logic chương trình mà ta phải nghĩ ở 1 thời điểm. Tương tự như trò tung hứng.
  + Ở cấp kiến trúc, để giảm sự phức tạp, ta có thể chia nhỏ hệ thống thành nhiều hệ thống nhỏ.
  + Mục tiêu khi giải quyết vấn đề (ko chỉ ở lập trình) là chia vấn đề phức tạp thành những phần đơn giản
  + **How to attack Complexity**
  + Sự phức tạp bắt nguồn từ:
    - 1 giải pháp phức tạp cho 1 vấn đề đơn giản
    - 1 giải pháp đơn giản (sai) cho 1 vấn đề phức tạp
    - 1 giải pháp không phù hợp, phức tạp cho 1 vấn đề phức tạp
  + Do cuộc sống càng ngày càng phức tạp,nên sự phức tạp trong lập trình là tất yếu.
  + 2 phương pháp để quản lý sự phức tạp:
    - Giảm sự phức tạp loại essential phải giải quyết ở 1 thời điểm
    - Đừng để phức tạp accidental sinh sôi
* **Desirable Characteristics Of a Design**
* 1 thiết kế tốt sẽ đạt đc những đặc tính sau đây, cái này có thể mâu thuẫn cái kia, nên cần sự cân bằng
  + **Giảm thiểu sự phức tạp**: tránh tạo ra những thiết kế độc đáo, thông minh, vì nó thường khó hiểu. Thay vào đó hãy tạo ra thiết kế đơn giản, dễ hiểu. Thiết kế phải cho phép ta có thể tập trung vào 1 phần mà ko cần quan tâm đến những phần #
  + **Dễ bảo trì**: Xây dựng chương trình mà khi 1 PG # đọc vào thấy dễ hiểu, ko cần doc
  + **Loose Coupling:** Giảm thiểu các sự kết nối giữa các phần trong hệ thống 1 cách tối đa
  + **Khả năng mở rộng:** mở rộng hệ thống mà ko va chạm, ảnh hưởng đến các kiến trúc nền tảng. Thay đổi 1 phần mà ko ảnh hưởng đến phần #.
  + **Khả năng sử dụng lại:** Sử dụng lại những phần khác
  + **High fan-in:** là số lượng các class dùng 1 class nào đó. Hệ thống nên có những class tiện ích tốt.
  + **Low-to-meium fan-out:** fan-out là số lượng class mà 1 class sử dụng. Số đó ko nên nhiều, nên nhỏ hơn 7, nếu nhiều thì sự phức tạp sẽ tăng lên.
  + **Khả năng di chuyển (portability):** Khả năng di chuyển hệ thống của mình qua 1 môi trường #.
  + **Leanness:** Thiết kế hệ thống mà ko có phần nào dư ra.
  + **Phân tầng (Stratification):** Phân tầng hệ thống trên sự phân tích, để ta có thể nhìn hệ thống ở 1 cấp độ mà ko quan tâm đến cấp độ khác. Ví dụ như ta viết 1 hệ thống mới dựa trên 1 hệ thống cũ chất lượng kém, ta có thể viết 1 lớp giao diện với code cũ, che đi phần cũ kém chất lượng đó.
  + **Kĩ thuật chuẩn:** Sử dụng những kĩ thuật thông dụng, dễ hiểu để ko làm hệ thống khó hiểu
* **Levels of Design**
* 1 số kỹ thuật thiết kế đc áp dụng cho mọi cấp độ của hệ thống, 1 số chỉ áp dụng ở 1 số cấp độ.



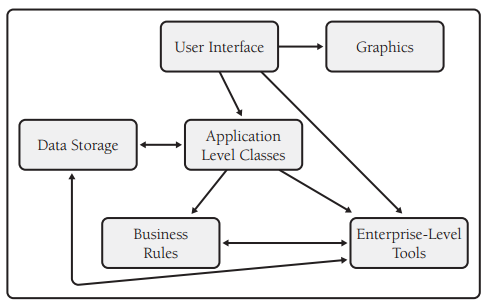
* + **Level 1: Software system**
  + **Level 2: Division into subsystems or packages**
  + Các hệ thống phụ có thể là DB, UI, business rules, phần báo cáo, log, …
  + Chia hệ thống thành những hệ thống phụ
  + Nếu tất cả hệ thống phụ có thể giao tiếp với tất cả hệ thống khác, thì ta mất đi lợi ích từ việc phân chia các hệ thống phụ, do đó cần hạn chế sự giao tiếp giữa các phần
  + Giả sử có 1 hệ thống



* + Nếu ta ko giới hạn sự giao tiếp giữa các khối, sự phức tạp sẽ tăng lên



* + Mỗi hệ thống phụ có thể giao tiếp trực tiếp với các hệ thống #, tạo ra những câu hỏi sau:
    - Khi thay đổi 1 thứ gì đó trong Graphics, có bao nhiêu phần khác mà PG cần phải quan tâm, hiểu.
    - Chuyện gì xảy ra khi sử dụng Business ở hệ thống #
    - Chuyện gì xảy ra nếu thêm 1 user interface mới.
    - Chuyện gì xảy ra nếu ta thay đổi lưu trữ dữ liệu bằng 1 máy ở chỗ khác
  + 1 phần giao tiếp với phần # chỉ khi nó thực sự cần biết về phần đó. Nếu ko chắc chắn thì đầu tiên nên hạn chế sự giao tiếp, sau đó nếu cần thì mới mở ra sự tương tác



* + Nguyên tắc khi thiết kế ở mức độ system: ko nên có quan hệ hình tròn, như class A dùng class B, class B dùng class C, class C dùng class A
  + **Những hệ thống thông dụng**: là những hệ thống thường xuất hiện trong các dự án.
    - **Business rules:** những qui tắc, luật, chính sách mà hệ thống phải thực hiện. Ví dụ nếu ta viết 1 hệ thống trả lương thì cần áp dụng các luật thuế, qui định tăng ca, ngày nghỉ, …
    - **User interface:** Tạo 1 hệ thống để tách UI ra để ko ảnh hưởng đến các phần #.
    - **Database access:** tập trung những thực hiện db lại 1 chỗ, giấu đi những chi tiết thực hiện db, cung cấp 1 lớp để giảm sự phức tạp của hệ thống
    - **System dependencies:** Đóng gói những thư viện vào 1 hệ thống riêng để có thể mở rộng. Ví dụ viết 1 hệ thống trên môi trường windows, đóng gói những lệnh gọi hàm của window vào 1 hệ thống riêng, như vậy sau này ta có thể thay đổi, ko dùng window nữa, mà có thể dùng linux.
  + **Level 3: Division into Classes**
  + Thiết kế ở mức độ này bao gồm xác định tất cả các class của hệ thống
  + **Classes vs Objects**
  + **Level 4: Division into Routines (Functions?)**
  + **Level 5: Internal Routine Design**
  + Gồm viết code giả, tra giải thuật, tổ chức code trong 1 hàm, và viết code

**Design Building Blocks: Heuristics**

* **Find real-world Objects**
* Phương pháp đầu tiên và phổ biến trong thiết kế là tìm các đối tượng thực tế, tạo các obj như thực tế, như obj person, obj employee, …
* Các thứ cần xác định khi thiết kế obj:
  + Xác định obj, các method, data
  + Obj có thể làm gì
  + Obj có thể làm gì với các obj #: gồm các việc: chứa các obj # và kế thừa
  + Quyết định phần nào của obj mà các obj # có thể và ko thể nhìn thấy
  + Định nghĩa interface: gồm public interface và protected interface
* Sau khi thiết kế xong thì lặp lại nhiều lần quá trình đó để đạt được kết quả tốt hơn
* **Form Consistent Abstractions**
* 1 class, 1 interface là 1 trừu tượng
* Sử dụng trừu tượng để giảm sự phức tạp, bỏ qua các chi tiết ko cần thiết
* **Encapsulate Implementation Details**
* Đóng gói giúp quản lý sự phức tạp bằng việc ko cho mình nhìn vào sự phức tạp
* **Inherit – When inheritance simplifies the design**
* Tính kế thừa là 1 trong những công cụ mạnh mẽ của lập trình hướng đối tượng. Nó sẽ tạo ra nhiều lợi ích nếu được sử dụng tốt, hoặc tạo ra nhiều thiệt hại nếu sử dụng ko đúng
* **Hide secrets**

---95

* **Class data mistaken for global data**
* Sử dụng biến toàn cục có 2 vấn đề:
  + 1 hàm sử dụng biến toàn cục ko biết có hàm khác cũng sử dụng biến đó
  + 1 hàm sử dụng biến toàn cục biết có hàm khác cũng sử dụng biến đó nhưng không biết là làm gì
* Khi 1 class có nhiều hàm hơn, class data sẽ giống như global data
* **Perceived performance penalties**
* Việc giấu thông tin sẽ làm cho việc truy xuất thông tin đó lâu hơn (qua nhiều hàm), làm giảm performance
* Do đó việc tăng performance sẽ cản trở việc giấu thông tin
* **Value of Information Hiding**
* Cần đặt câu hỏi: Cái nào cần phải giấu đi?

**Identify Areas Likely to Change**

* Các bước:
* 1. Xác định những thứ có khả năng thay đổi
* 2. Đưa nó vào những class riêng
* 3. Cô lập những class đó: tạo những interface trung gian, cho dù có thay đổi thì những class ngoài sử dụng cũng ko cần quan tâm
* 1 số vùng có thường khả năng thay đổi: business rules, hardware dependencies, I/O, nonstandard language features, difficult designs, status variables, data-size constraints.
* Status variables:
  + Đừng sử dụng kiểu bool, mà dùng kiểu enum. Vì ta rất dễ thêm trạng thái vào biến.
  + Ko truy xuất trực tiếp vào biến mà dùng hàm
* **Anticipating Different Degrees Of Change**

**Keep Coupling Loose**

* Sự kết nối, liên quan giữa các class, các hàm
* Good coupling: sự kết nối giữa các khối là tốt nếu 1 khối có thể đc sử dụng dễ dàng bởi các khối #.
* Làm cho các kết nối đơn giản nhất có thể
* Cái mà nó cần biết
* Sin() vs sin(var1, var2, var3, …): cái thứ 1 có kết nối lỏng hơn vì các class ở ngoài ko pik rõ bên trong
* **Coupling Criteria**
* Các tiêu chuẩn để đánh giá:
  + Size: interface càng nhỏ thì càng dễ kết nối. 1 hàm nhận 1 tham số sẽ dễ dùng hơn 1 hàm nhận 6 tham số
  + Visibility:

--- 100

--- 100

* + Flexibility: Khả năng thay đổi kết nối giữa các module
* **Kind of Coupling**
* 1 số loại kết nối phổ biến:
  + Simple-data-parameter coupling: dữ liệu truyền giữa 2 khối là dữ liệu primitive, thông qua danh sách tham số. Kiểu kết nối này bình thường và có thể chấp nhận
  + Simple-object coupling: 1 module kết nối với 1 obj = cách khởi tạo obj đó. Kiểu kết nối này ổn.
  + Object-parameter coupling: Obj1 yêu cầu Obj2 truyền cho nó Obj3. Kiểu này chặt hơn vì Obj2 phải biết Obj3, thay vì chỉ truyền các dữ liệu primitive
  + Semantic Coupling: Đây là kiểu kết nối phức tạp, nguy hiểm, khi 1 module ko chỉ dùng các thành phần cú pháp của module #, mà còn sử dụng thông tin ngữ nghĩa bên trong module ấy
    - Module1 đưa 1 flag cho module2 để làm 1 việc j đó, điều này có thể yêu cầu module2 biết về logic trong module1. Nếu flag có kiểu là (enum, obj) thì kiểu này ổn
    - Module2 sử dụng biến toàn cục sau khi module1 đã sử dụng -> m2 phải biết m1 đã chỉnh sửa biến toàn cục ntn
    - M1 qui định: để gọi M1.a() thì phải gọi M1.b() trc, nhưng M2 biết khi gọi M1.a() thì nó sẽ tự gọi M1.b(), nên M2 tạo M1 và gọi luôn M1.a()

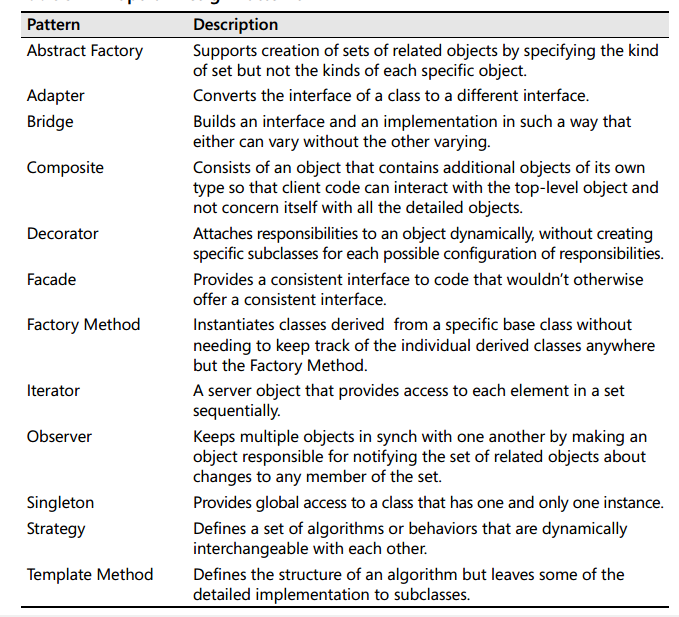
--- 102

---102

* + - M1 truyền obj cho M2. Vì M1 biết M2 chỉ dùng 3 phương thức của Obj nên M1 tạo Obj với 1 số data cần thiết
    - M1 truyền obj1 cho M2. M2 biết thực chất M1 truyền cho nó obj2 nên M2 chuyển obj1 thành obj2 và gọi những phương thức của obj2
  + Để tạo ra kết nối linh hoạt, module cần tạo ra 1 lớp trừu tượng, cho phép ta có thể tập trung vào 1 thứ tại 1 thời điểm.
  + Khi sử dụng 1 module #, mà ta phải quan tâm tới logic bên trong module đó, chỉnh sửa biến toàn cục, … thì kết nối đó ko tốt.
  + Hàm, class là những công cụ đầu tiên để làm giảm sự phức tạp

**Look for Common Design Patterns**

* **Patterns reduce complexity by providing ready-made abstractions**
* **Patterns reduce errors by institutionalizing details of common solutions**
* **Patterns provide heuristic value by suggesting design alternatives**
* **Patterns streamline communication by moving the design dialog to a higher level**



**Other Heuristics**

--- 105

--- 125

**Chapter 6: Working Class**

Class là 1 công cụ để giảm sự phức tạp, cho phép ở 1 thời điểm, ta có thể tập trung vào 1 phần mà ko cần quan tâm đến những phần khác

**Class Foundations: Abstract Data Types (ADTs)**

* ADT là 1 bộ các dữ liệu và các hoạt động trên các dữ liệu đó.
* Các hoạt động sẽ mô tả dữ liệu cho phần còn lại của hệ thống và cho phép hệ thống thay đổi dữ liệu đó
* Dữ liệu trong ADT có nghĩa rộng
* Khi thiết kế, phải nghĩ về ADT trc khi nghĩ về class

--- 126