REVIEW SÁCH CHƯƠNG 8:
TÍNH TÁI SỬ DỤNG VÀ KHẢ NĂNG DI ĐỘNG

I. Tóm tắt nội dung:

Chương này thảo luận về **tái sử dụng phần mềm (reusability)** và **khả năng di động (portability)** – hai yếu tố quan trọng trong phát triển phần mềm hiện đại. Việc tái sử dụng giúp giảm thời gian phát triển, trong khi khả năng di động đảm bảo phần mềm có thể chạy trên nhiều nền tảng khác nhau mà không cần thay đổi đáng kể.

1. Tái sử dụng phần mềm (Software Reusability):

Tái sử dụng là việc dùng lại các thành phần của một sản phẩm phần mềm trong một sản phẩm khác, giúp tăng tốc độ phát triển và giảm lỗi.

- Các loại tái sử dụng phần mềm:

- **Tái sử dụng cơ hội (Opportunistic Reuse)**: Khi một nhóm phát triển nhận ra rằng họ có thể dùng lại một phần mềm đã có trước đó
- **Tái sử dụng có hệ thống (Systematic Reuse)**: Khi các thành phần phần mềm được thiết kế sẵn để có thể sử dụng lại trong tương lai.

- Lợi ích của tái sử dụng:

- **Giảm thời gian phát triển**: Không cần viết lại từ đầu.
- **Cải thiện chất lượng**: Thành phần đã được kiểm thử trước đó.
- **Giảm chi phí bảo trì**: Phần mềm chuẩn hóa dễ bảo trì hơn.

- Trở ngại của tái sử dụng:

- **Hội chứng "Không phải do tôi làm" (Not Invented Here - NIH)**: Lập trình viên có xu hướng không tin tưởng vào mã do người khác viết.
- **Vấn đề lưu trữ và tìm kiếm thành phần tái sử dụng**: Cần có hệ thống quản lý thư viện mã hiệu quả.
- **Chi phí phát triển thành phần tái sử dụng**: Viết một module có thể tái sử dụng đòi hỏi công sức lớn hơn so với một module đơn lẻ.
- **Vấn đề pháp lý**: Khi tái sử dụng phần mềm do công ty khác phát triển.

- Ví dụ thực tế về tái sử dụng:

- **Raytheon Missile Systems Division**: Một công ty phát triển hệ thống tên lửa đã đạt được mức tái sử dụng 60%, giúp tăng năng suất lên 50%.
- **Sự cố Ariane 5 (1996)**: Một tên lửa của Cơ quan Vũ trụ Châu Âu phát nổ do một lỗi trong phần mềm tái sử dụng từ Ariane 4 mà không được kiểm thử lại.

---

2. Khả năng di động của phần mềm (Software Portability):

Phần mềm có **khả năng di động** nếu nó có thể dễ dàng chạy trên các nền tảng khác nhau mà không cần viết lại từ đầu.

- Các vấn đề ảnh hưởng đến tính di động:

- **Khác biệt phần cứng**: Bộ xử lý, bộ nhớ, kiến trúc máy tính khác nhau có thể gây lỗi khi chạy chương trình.
- **Hệ điều hành khác nhau**: Ví dụ, Windows và Linux có cách xử lý file, bộ nhớ khác nhau.
- **Trình biên dịch (Compiler)**: Cùng một mã nguồn nhưng có thể biên dịch khác nhau trên các nền tảng.

- Cách tăng cường tính di động:

- **Sử dụng ngôn ngữ lập trình độc lập với nền tảng** (Java, Python)
- **Viết mã theo tiêu chuẩn chung** (POSIX, ANSI C).
- **Sử dụng thư viện hỗ trợ di động** (Qt cho GUI, OpenGL cho đồ họa).
- **Tận dụng máy ảo (Virtual Machines)** để chạy mã trên nhiều hệ thống.

---

3. Các phương pháp được sử dụng trong chương:

- **Phân tích các case study** để minh họa lợi ích và thách thức của tái sử dụng và di động
- **Phương pháp thiết kế mẫu (Design Patterns)** giúp tạo ra các cấu trúc phần mềm có thể tái sử dụng
- **Tích hợp thư viện và API** để hỗ trợ tính di động.

II. Bài học rút ra:

- Tái sử dụng giúp tiết kiệm thời gian, nhưng cần kiểm thử lại khi sử dụng trong môi trường khác nhau.
- Thiết kế phần mềm tốt ngay từ đầu giúp tăng khả năng tái sử dụng và di động.
- Sử dụng công nghệ phù hợp (như ngôn ngữ độc lập nền tảng) giúp giảm rủi ro khi chuyển đổi phần mềm.
- Việc quản lý thư viện mã và tài sản phần mềm là chìa khóa để tối ưu hóa tái sử dụng.

=> **Tóm lại**, tái sử dụng và khả năng di động là hai yếu tố quan trọng giúp phần mềm linh hoạt hơn, giảm chi phí và cải thiện chất lượng!