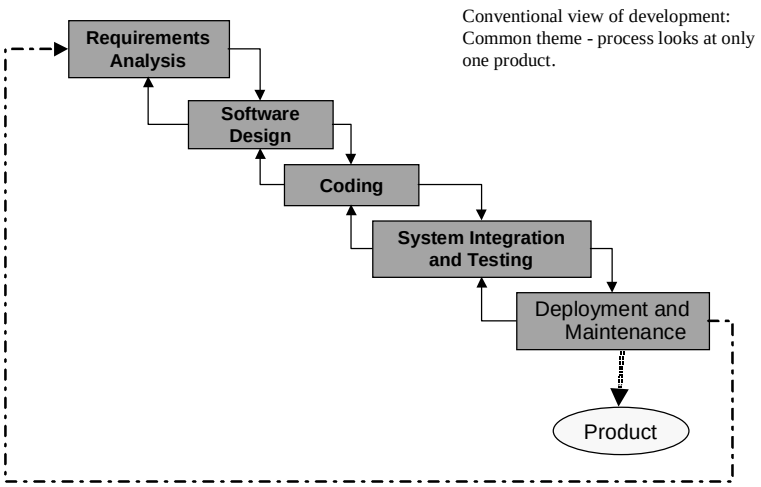
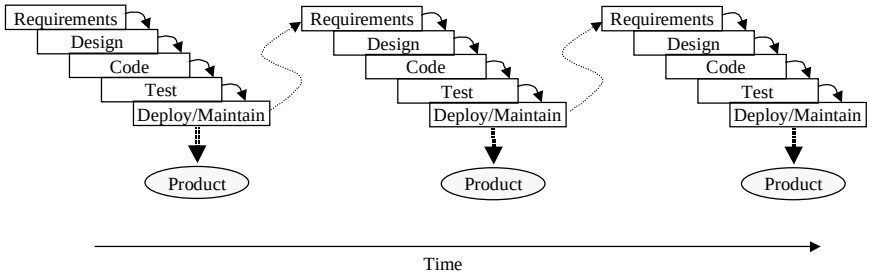
Software Reuse

From informal reuse (scavenging)
to systematic reuse
Management and technical issues

Merry-Go-Round of Sequential Development



Sequential Development Over Time



Inefficiencies of Sequential Development

- Observation: much of software "development" is really re-development.
  - Software inevitably exists in many versions
  - Seldom develop truly new applications
- Implication: typically much in common among systems we build …But very little is reused
  - Difficult to identify commonalties and differences
  - Difficult to reuse code components
  - Difficult to add desired feature to existing design
  - Difficult to adapt other work products (if they exist)
  - Generally easier to re-do than re-use
- Result: enduring "software crisis"
  - Software remains a fundamentally hand-crafted (as opposed to manufactured) product
  - Only small improvements to schedule, cost, quality are possible

Motivations

- Development cost
  - it is (or should be) cheaper to use existing software components than to develop them “from scratch”
  - cost advantage is not only for code: also for specifications, design, test, documentation
- Cycle time
  - adapting existing software should be faster than writing new software
- Predictability
  - reuse and adaptation should not only be faster, but should also be easier to predict

Stage 0: Scavenging

- Code scavenging: Use existing component as “template” for new component
  - New code (or document, or ...) is constructed by editing an existing file which is “close” or has at least some common parts
- Almost universal for code. Very few components begin as empty files
- Often completely ad hoc and personal

Limitations of ad hoc scavenging

- Time savings are limited to initial coding
  - Only code (not documents, not test cases, ...) are reused
  - Changes (editing) is arbitrary, so there is no savings in test effort
- Maintenance problems
  - Fixes and enhances must be applied to each copy of reused code

Stages of Reuse

- Stage 0: Scavenging
- Stage 0.5: Template Libraries
- Stage 1: Component Reuse
- Stage 1.5: Component Frameworks
- Stage 2: Higher Level Programming
- Stage 3: Systematic Reuse

Stage 0.5: Template libraries

- Organizational support for reuse
  - Maintain a library of “template” modules
    - Shared and classified for efficient location
  - May include quality control (approved templates)
  - Should include record keeping and traceability
    - How many times was this template reused last year?
    - Which modules are based on it?
- Still limited
  - Maintaining several variants is still expensive
  - Inspections, testing, user documentation etc. may be accelerated but not fully reused

Stage 1: Component reuse

- Better to re-use a component *without change*
  - Reuse testing, inspection, documentation, etc., not only coding effort
  - Component dependability improves with reuse
  - Maintain and enhance one version
- Component library is an organizational asset
  - Maintaining and enhancing it is an investment

Barriers to Component Reuse

- Organizational and contractual
  - Customers (e.g., U.S. D.o.D.) who want to pay only for "new development"
  - Organizations that measure productivity by amount of new code written
  - Budgeting extra effort to produce general, reusable components (typically 2x or 3x cost of single-use component)
- Technical
  - Finding, understanding, assessing, and "fitting" components

Finding Reusable Components

- Partial match problem
  - There is seldom a component that does *exactly* what is needed; we look for components that do *most* of or *almost* what is needed
    - Example: Search the web for a "best bus route" component, or parts. What do you look for?
- Sipping from the firehose (information overload)
  - There are often *too many* components that do most of or almost what we need.
  - Many are not really suitable; it is easy to lose the few that are.

Understanding Reusable Components

- Large libraries are complex
  - Example: Leda graph structures/algorithms library
    - Possibly no savings in the first use
  - Example: Motif user interface toolkit (or Mac toolbox, or Windows API, or ...)
- Documentation is essential
  - Orientation to the library as a whole
  - Indexing and organization to find what is needed
  - Clear, complete descriptions of components and (especially) component dependencies
  - Complete examples (templates again?) are helpful

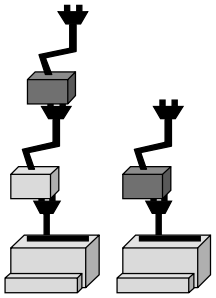
Assessing Reusable Components

- Does this component do what I need?
- Is it dependable?
- Is it (small | fast) enough?
- Does it fit?

Component Mismatch

- Analogy: My printer
  - The printer is just fine — with 110v AC current, 50Hz
  - In Italy it is useless
- Software component mismatches
  - Wrong programming language
  - Wrong interface
    - file io vs. procedure arguments
    - data push vs. data pull, internal vs. external control
  - Wrong assumptions
    - shared vs. copied structures
    - error handling

Fitting Reusable Components



- A mismatch may not be fatal; we may be able to adapt to a component
- Often there is more than one strategy
  - Analogy: Adapt 220v to 110v for my printer, or replace the transformer?
  - Similar in circuit design: “glue logic” fits standard ICs to their roles in the overall circuit
- Approaches
  - Portability layer (for whole library), shims
  - Wrappers, servers (for language & interface mismatch)

Stage 1.5: Component Frameworks

- Organized component libraries with standard "patterns" of use
  - Patterns may be templates
  - Clear overall principles of organization
  - Inheritance may help organize library of OO framework
- Examples (for user interface)
  - MetroWerks PowerPlant; Microsoft Foundation Classes; SmallTalk MVC

Investing in a Framework

- Wide scope frameworks are usually cheaper to buy than to build
  - Examples: The interface/application frameworks on previous slide; domain-specific frameworks for accounting, real-time control, simulation, etc.
  - Narrow domain frameworks can be developed gradually over time
  - Accumulate, refine, organize: Not one big investment, but an ongoing effort to build a foundation for future development

Stage 2: Higher level programming

- There is no clear line between library and language
  - Intermediate stage (1.75?) is partial generation of applications using a framework (e.g., interface "painters")
- Eventually a domain becomes "formalized"
  - Standard notation and semantics with corresponding component support for "programming" at the domain level
  - Closely related to (domain-specific) software architectures and virtual machines
  - Example: SQL has (mostly) replaced lower-level programming of database functions

From here to there ...

- It is probably *not* possible to jump from ad hoc reuse to a framework in one step
  - Premature efforts to build reusable components are usually wasted
- Incremental strategy
  - Use ad hoc reuse to trigger reusable component construction:
    - Can I retrofit a generalized component to its original context *and* the new context?
  - Use maintenance history to identify the "right" component interfaces:
    - Can I factor the rapidly changing parts from the stable or slowly evolving parts (e.g., with a mechanism/policy split)?

Management Support for Reuse

- Remove obstacles
  - Reward system and corporate culture must place as high (or higher) value on reusing and improving, as on producing entirely new software
  - Mistake to avoid: rewarding production of "reusable" components more than actual reuse
- Organize and make visible
  - Make identification, assessment, and adaptation of reusable parts an explicit part of development
  - Include feedback mechanisms
- Provide adequate support
  - Budget extra effort to improve the asset
  - *BUT* move incrementally — avoid a disasterous big-bang effort

Summary — Reuse

- More than just faster coding
  - Goal is reuse of design, documentation, test and analysis, etc., and reduction of maintenance effort, in addition to faster production of software
  - The situation is not so bad
  - Commercial component frameworks are reuse successes on a grand scale (but often ignored as such)
  - But it could be better ... at the domain & organization level
- Some issues are non-technical
  - Management and organization support are essential
- Reuse can be approached incrementally
  - Gradually move from ad hoc reuse to component libraries, frameworks, and domain engines

Product-Line Engineering for Software

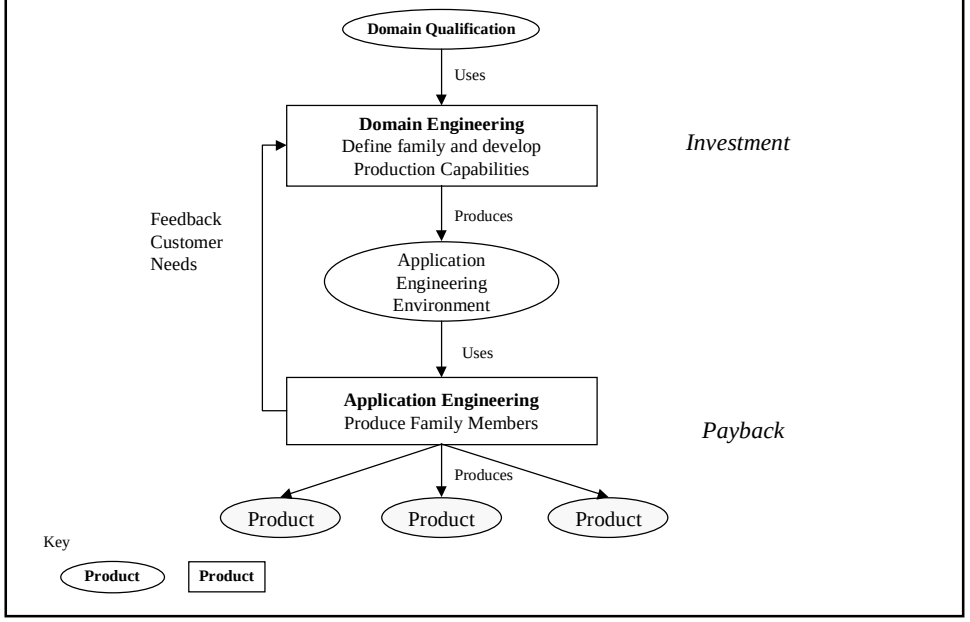
Domain Analysis

Domain Engineering

Approach

- Integrate development process and product
  - Design for (re)producibility
  - Concurrent engineering for software (process and product are designed together)
- Reorganize development process
  - Evolve a family rather than build single systems
  - Production environment as product
- Systematic approach to building flexible *application generators*
- Can be done with existing technology

P/L Development Process



Idealized P/L Development Process