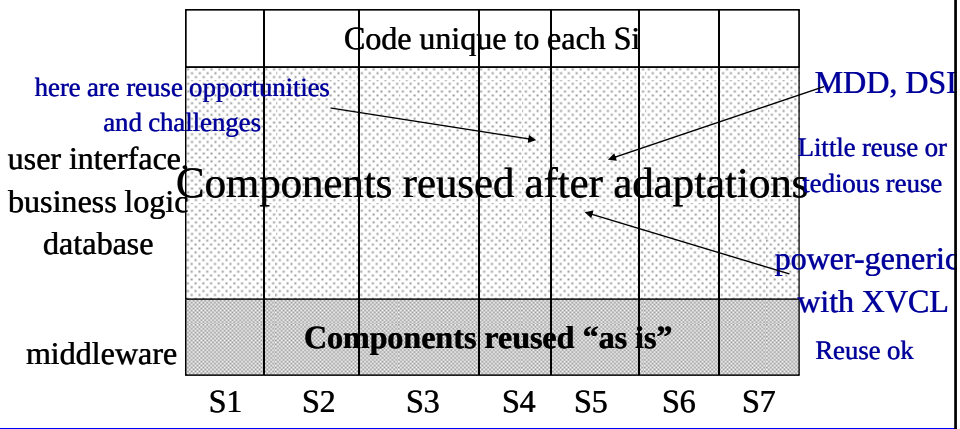
CS6201 Software Reuse

Lecture Notes Set# 4: More on Reuse and PLs

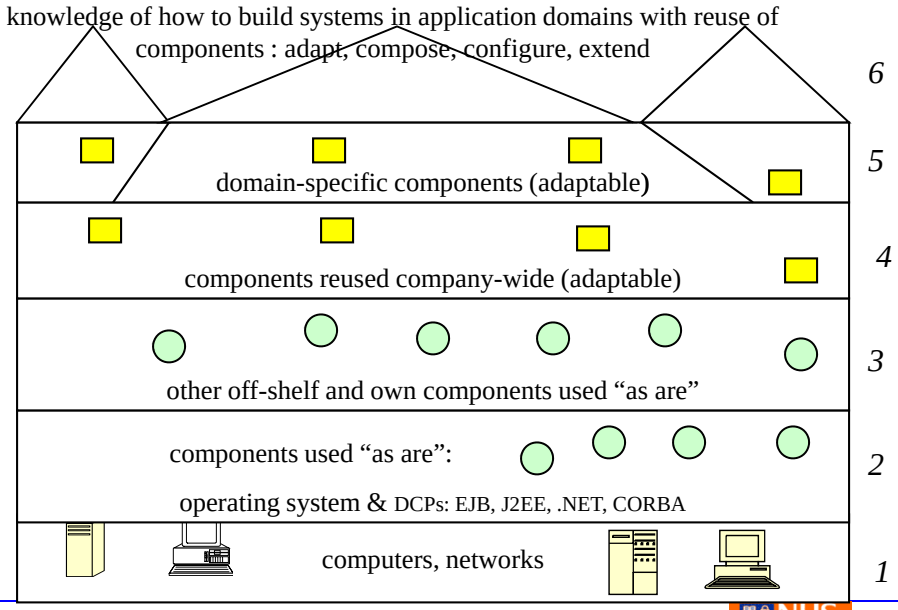
1. Reuse: general observations, reuse stats
2. How companies realize PL approach?
3. Comments on SOA and PLs
4. Many meanings of "software architecture"
5. Reuse: hardware vs. software

Simalarity and reuse problem

- Systems Si evolved from the same original system
  - They are all similar to each other but also different



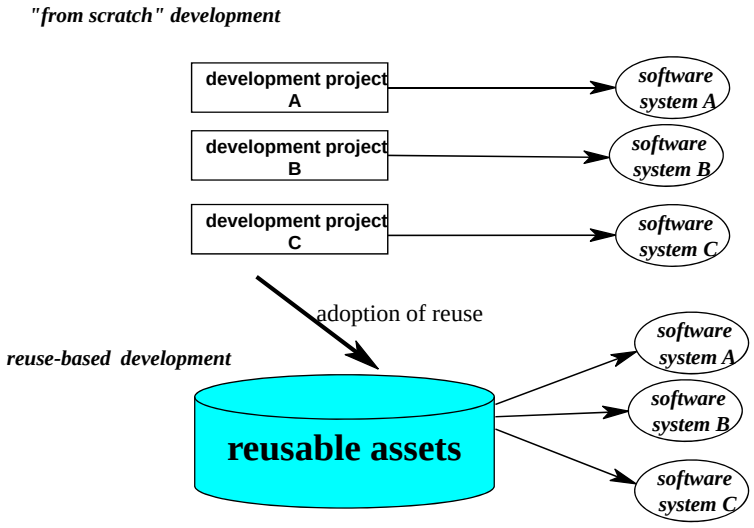
Levels of reuse infrastructure



Objectives of reuse and PL

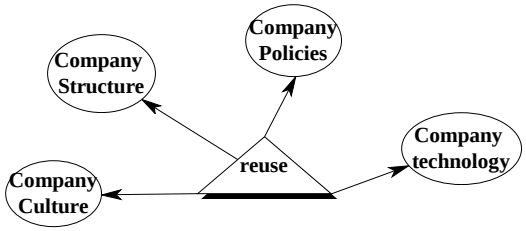
- Reduction of the product development cost
- Reduction of time-to-the-market
  - Expanding the range of products to address new customers or market segments
- Reduction of maintenance

Transition to reuse-based development



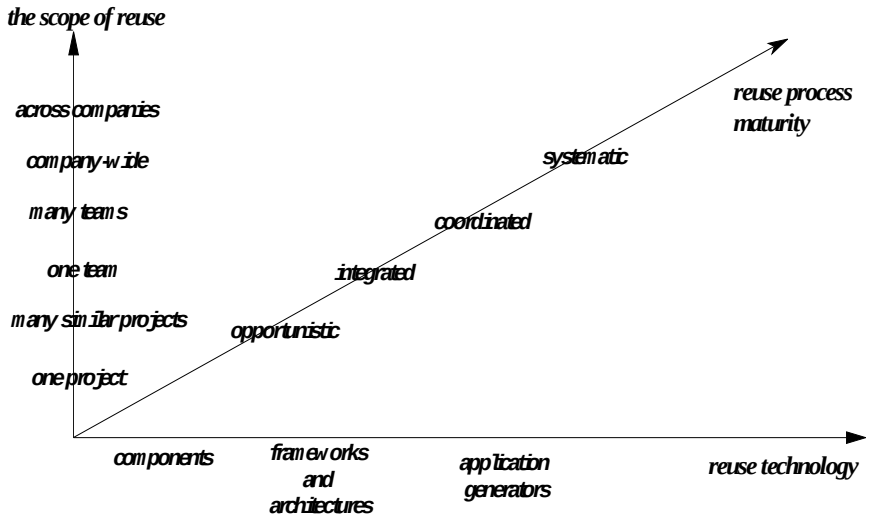
Reuse affects a company

- reuse requires changes in a company:
  - culture: develop for others and use others work
  - policies: setting up reuse procedures, reward system, monitoring reuse
  - structure: domain engineers and product developers
  - technology: reuse methods and tool

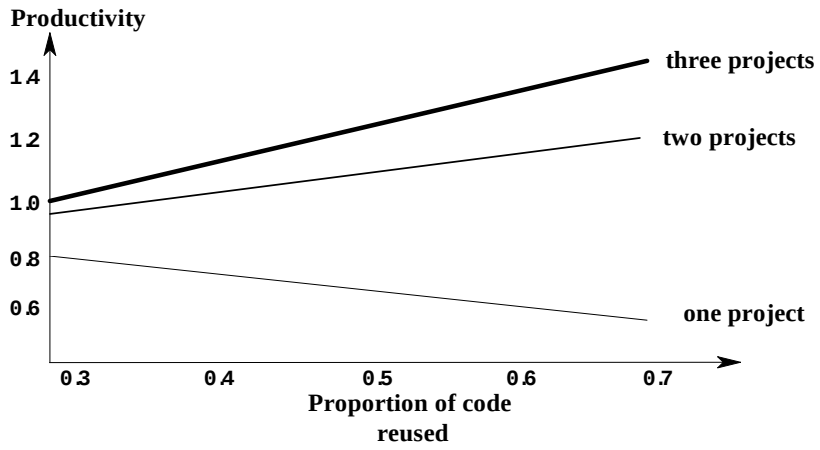


- reuse can only be implemented incrementally

Maturity of reuse practice



How reuse scope affects reuse benefit



Some reuse statistics (old)

- Hitachi: reduced number of late projects from 72% to 7% in 4 years
- Toshiba: improved productivity 3 times in 9 years, 50% code reuse
- Toshiba: reduced error rate from 7-20 per 1 KLOC to 2-3
- Fujitsu: improved productivity by 2/3, reduced error rate by factor of 10
- NEC: increased productivity by 26% to 91%
- NobelTech: doubled productivity
- HP: shortened time-to-market by factor of 4, reduced error rate by factor of 10
- frame technology (Netron): up to 90% reuse(Bassett 97)

Frames in industry

Frame Technology (by Netron, Inc)

- Applied to large business systems in COBOL
- Productivity indicators based on assessment by QSM:
  - "time-to-market reduction by 70%"
  - "project costs reduction by 84%"
  - "reuse percentage from 50% to 95%"

Details in: Bassett, P. Framing software reuse - lessons from real world, Yourdon Press, Prentice Hall, 1997

XVCL in industry

STEE experience: Web Portals in ASP/XVCL

- Over 20 different portals built/maintained with ASP/XVCL
- Short time (less than 2 weeks) and small effort (2 persons) to start seeing the benefits
- Development productivity indicators:
  - 60% - 90% reduction of code needed to build a new portal
  - estimated eight-fold reduction of development effort
- Maintenance productivity indicators:
  - for the for first nine portals, managed code lines was 22% less than the original single portal

Retive Solutions PTe Ltd: CMRS-PL in JEE/XVCL

- On-going project; objective: technology transfer

What impedes reuse?

- Technology factors
  - There is nothing to reuse
  - Software component is too inefficient for a task in hand
  - Software component is too specialized for a task in hand
  - Hard to modify: a software component does not do exactly what we want but it is difficult to modify it
  - Hard to integrate a software component with the rest of the system
  - The cost of finding, changing and testing of a software component is bigger than the cost of writing anew
  - Poor software structure - programmers do not understand a software component

What impedes reuse?

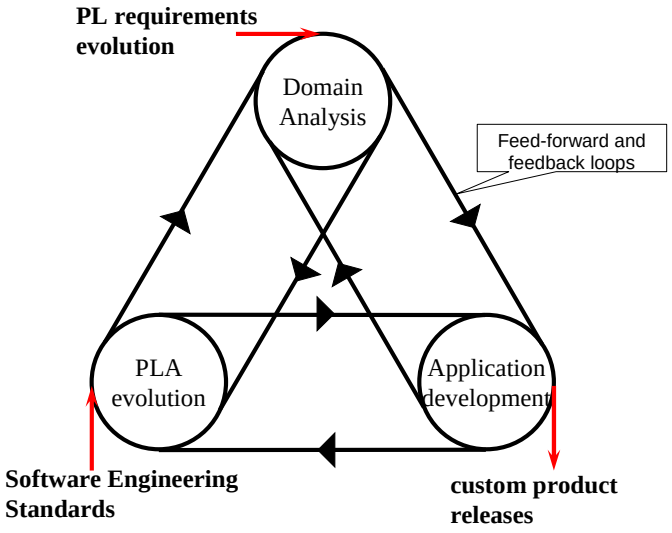
- Software documentation:
- Lack of requirement/design/code documentation - programmers do not understand a software component
- Inconsistent, ambiguous and incomplete documentation: we cannot determine what a given software component does without examining the code

- Psychological factors:
  - Not Invented Here syndrome
  - It is more fun to write software anew rather than to reuse.
  - Reusing may mean that I cannot do this myself.

What impedes reuse?

- Organizational and managerial factors:
  - failing to establish reuse-oriented company policies and infrastructure
    - no incentives for writing reusable software and for reusing software
  - failing to measure and demonstrate the benefits of reuse; high initial cost of reuse programme
  - lack of commitment and support for reuse programme from high management
  - failing to cope with company changes triggered by reuse
  - not providing enough training

PL Processes



How companies realize PLs?

What is PL?

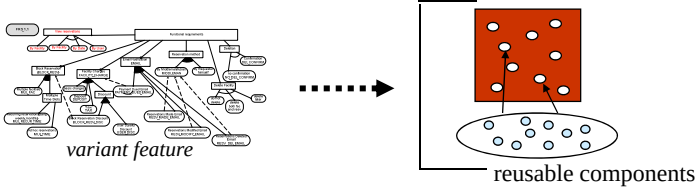
- A software product line is, fundamentally, a set of related products. Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture under the auspices of a production plan. New or updated core assets are rolled back into the core asset base for future systems.
- P. Clements on PL, SEI: http://www.sei.cmu.edu/news-at-sei/columns/software-product-lines/software-product-lines.htm

Two types of PLs

- Fine-granularity components in PLA
  - Many components reused in each product
  - PLA: Component versions from past products
  - Complex inter-component dependencies
  - Example: Bosch PL
- Large-granularity components in PLA
  - Smaller number of large components in PLA
  - Standardization, documentation, process
  - Educating staff
  - Example: Tektronix PL

Product derivation: Bosch

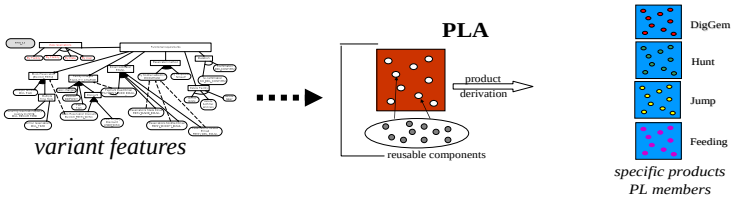
1. **Analyze requirements** for new product: select variant features



2. ***Initial phase*:**
   a) understand the impact of variant features on components
   b) select component configurations that "best match" new product
3. ***Iteration phase*:**
   a) adapt selected components, replace/add yet other components
   b) integrate components, validate the new product

Comments on Bosch PL

- the impact of variant features spreads through many components!

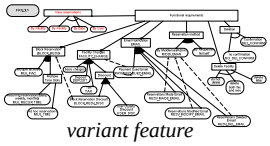


- explosion of look-alike component versions
  - same functionality implemented in variant forms, in hundreds of similar component versions
- complex, hidden dependencies among reusable components

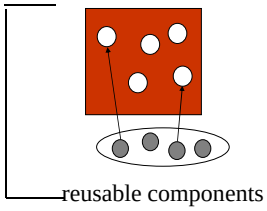
➢ how do I reuse already implemented functionality?
  - selecting and adapting component configurations for reuse

Product derivation: Tektronix

1. **Analyze requirements** for new product: select variant features



variant feature



reusable components

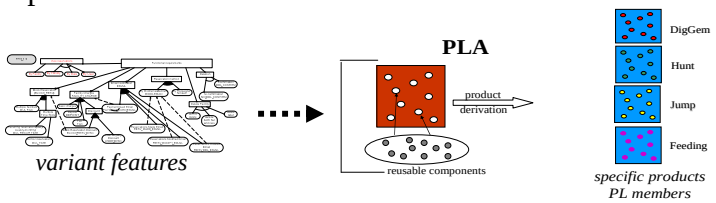
2. **Understand the impact of variant features on components**
3. **Iteration phase:**
   a) adapt selected components, replace/add yet other components
   b) integrate components, validate the new product

Comments on Tektronix PL

- techniques for component generalization:
  - cpp, configuration parameters
- not much global controls to streamline customizations across software assets
- little automation during product derivation
  - wizards and GUIs for customization during product

Reuse problems (general)

- Complex and invisible impact of variant features on components



- Much manual work during product derivation
  - For given variant features – which components should I customize and how?
  - Not much global controls and automation to streamline customizations across software assets
- Difficulty to reuse of already implemented functionality

Selecting and scoping a PL

- Selecting a PL is driven by business considerations
  - there must be a business value in a PL:
    - the profits must outweigh the investment in reuse
    - many customers requesting different variants of a system
    - savings in development cost, time-to-the-market
- Scoping a PL:
  - what should we engineer for reuse in a given PL?
    - functional variants?
    - portability across a range of platforms?

Building and evolving PLA

- Proactive approach – domain engineering
  - trying to anticipate variants (domain analysis)
  - design pf a PL architecture to cater for variants
- Extractive approach
  - extract features from existing system(s)
  - design PL architecture based on that
- Reactive approach (iterative)
  - add new variants as they appear in systems built for various customers
  - refine a PL architecture with variants as they come

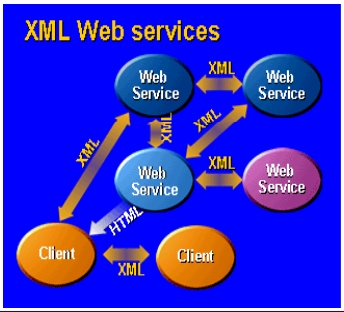
Techniques for reuse

1. libraries of functions (sine, sqrt, etc.)
2. macros, pre-processors (cpp)
3. code generators, 4GL, CASE, IDE, GUI
4. parameterization: generics (Java, C#), templates (C++), …
5. OO approach, design patterns, OO frameworks
6. software architectures and component-based approaches
7. platforms: J2EE, .NET
8. ERP packages such as SAP, PeopleTools
   - accounting, payroll, customer order processing
9. Generative techniques such as XVCL

SOA, Web services

Service-Oriented Architecture (SOA)

- Web applications built out of loosely-integrated *services* distributed over WWW
- Each service performs some business function
  - a credit checking service
  - a stock quote service
  - a purchasing service
- Same services can be combined together to create many similar applications (reuse)



Software Product Line (SPL)

- A family of similar software products that satisfy needs of a particular customer group
- These products are managed from a common, reusable base of SPL core assets

Feature management challenge

- One feature may affect many product components

Features interactions:

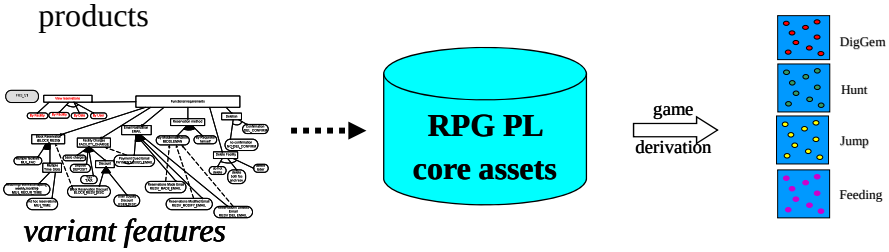
- Functionally interdependent features:
  - If I select one feature I must also select some other features
- One feature may affect implementation of other features

Types of features

- Coarse-grained feature: implemented in source files that are included into a customized product when feature is selected
- Fine-grained feature: affects many product components, at many variation points
- Mixed-grained feature: involves both fine- and coarse-grained impacts on components

Steps towards RPG SPL

- A key to reuse are flexible, adaptable components
- Design architecture and reusable components for RPGs
  - Apply extra variation mechanisms for component adaptation
- Static adaptation and configuration of components to build custom products



SOA and SPL

- High-level goal is the same for both SOA and SPL:
  - Cost-effective engineering of similar applications
  - Apply reuse for rapid development of applications
- Technical challenges are also similar:
  - Component/service description (reuse)
  - Adaptation of components/services
  - Flexible composition and reconfiguration of components/services
  - Architectures/workflows
  - Variation mechanisms

We now examine closer similarities and differences between SOA and conventional SPLs

SOA vs. conventional SPL

- Service-based products form SPL on SOA
- Services vs. components

SOA service	SPL component
Service implements well-defined business function	Component may be just any building block for products
Service description : WSDL, ontologies must describe service advertisement, discovery; service quality	Component description: API; parameters
Service orchestration	Use of architectures
Third part services	In-house (and third party) components

Dynamic vs. static configuration

- Service based products must be customizable, re-configurable (at runtime)
- A conventional SPL, typically relies on static customization (at design time)

dynamic	static

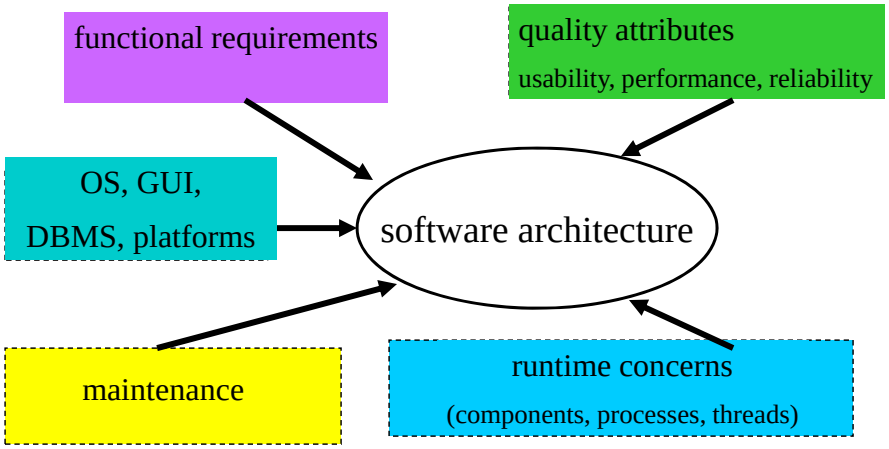
Software Architecture may mean many things …

Goals of software architectures

- to achieve uniformity across products
- to improve software productivity and quality
- to prevent programs from decaying
- to facilitate communication between stakeholders:
  - business and technical people
  - users and developers
- what constitutes an architecture depends on the perspective of a given stakeholder
- different perspectives yield different architectures

Software architecture

architecture must satisfy concerns of different stakeholders:



Examples of architectural views:

- a framework for satisfying requirements:
  - early evaluation of critical system requirements (functional, performance, etc.)
  - traceability from requirements to code
  - rationale for design decisions
  - supporting answering "what-if" questions
- a basis for partitioning the system:
  - into runtime architecture, components, subsystems
  - into reusable software building blocks
  - to enable project management: planning and estimation
  - to assign tasks to project team members

Software architecture – definition:

an abstract view of a system structure in terms of its:

- components,
- component properties, and
- component relationships

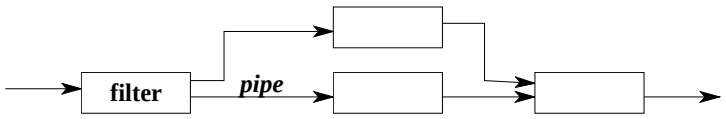
What is a "component", "property" and "relationship" depends on the view and the intended role of the architecture.

- levels of abstraction:
  - from physical architecture to conceptual architecture

Architectural styles

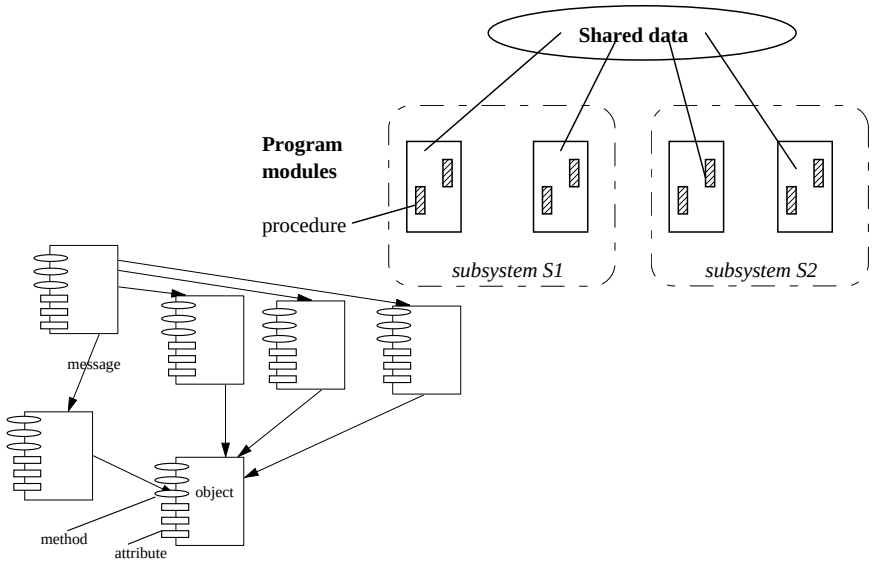
- typical ways to organize runtime architecture:
  - pipes & filters
  - procedural
  - OO
  - layered
  - client-server
  - distributed
  - event-based implicit invocation

Pipes & filters

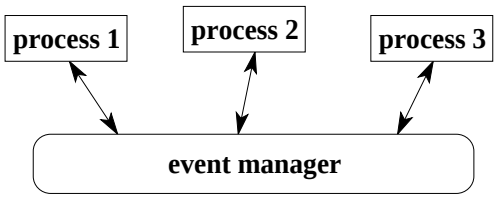


- *filters* are components: read a stream of data on input, transform data and produce a stream of transformed data on output
- *pipes* are connectors: transmit data among filters

Procedural and OO styles



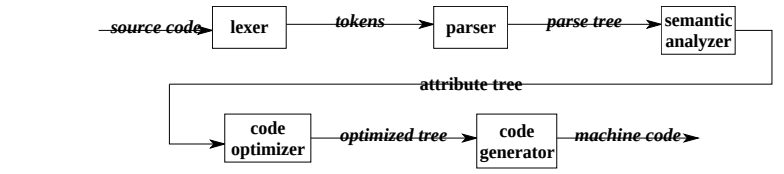
Event-based, implicit invocation style

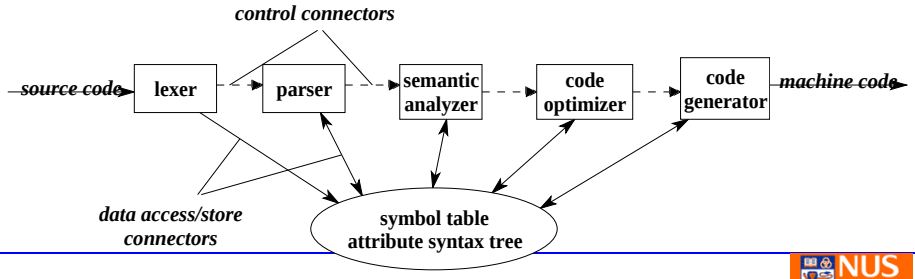


- processes announce events that mark completion of some task
- other processes can register their interest in specific events
- once an event occurs, an event manager invokes processes that registered interest in that event

Reference architecture

- structure of a solution in a domain
  - compiler: parser, semantic analyzer, optimizer and code generator





Runtime vs. design-time architecture views

software runtime architecture view:

- defines physical or logical software system structure at runtime:
  - components, component interfaces, subsystems, protocols

software design-time architecture view:

- serves the purpose of cost-effective development and maintenance of software systems
  - reusable components, impact of change

Runtime architecture view:

- platforms
- runtime system components
- connectors
- deployment of components
- synchronization
- control

Design-time architecture view:

- who does what?
- development & maintenance of system components
- how to implement new requirement or fix a bug?
  - which components should I revise?
  - which components should I test?
- if I change component x, which other components may be affected?
- portability, reusability, ease of maintenance

Examples of design-time architectures

- program instrumented with macros
- OO framework
- x-framework
- generators
- GUI
- IDE
- mechanisms of JEE or .NET
- other examples?

Reuse: hardware vs. software

Reuse in other engineering disciplines

- is each product model designed from scratch?
  - car industry
  - house construction – standardized doors, prefabricated walls
  - computer industry – PCs built of integrated circuits

what is the essence of reuse in above examples?

can we do the same in software?

- is software like hardware?
- do approaches that work in classical engineering also work for software?
  - objects were proclaimed software ICs
  - Java Bean, .NET and CORBA components

Hardware design and production

evolution of car models

- distinction between product design vs. product manufacturing
- evolution (reuse?) of the design, but
- replacement - not modification - of physical products

Software production process

software production phases:

requirements analysis

architectural design

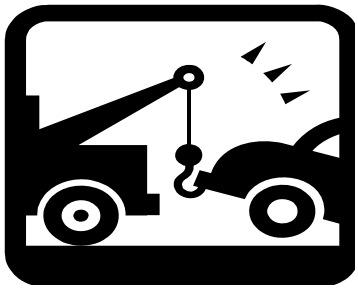
implementation

testing

maintenance

- iterations across phases, changes of product architecture
- changes of requirements during the project
- lack of evaluation at the level of the architecture
- low level of reuse
- one architecture --> one program

hardware wares out



hardware cannot be copied for free

Hardware vs. software maintenance

hardware:

- parts wear out, need fixing or replacement

software:

- parts do not wear out
- drastic and frequent changes, enhancements (50%-80%)
  - business environment and computer/software technology
  - software must be very flexible to stay in sync with evolving environment
  - still – software is hard to change
- software decays during maintenance
- successful software may be around many years

Hardware – software analogy?

- despite similarities, not completely so:

hardware	**software**
strict physical boundaries	product of thought
constrained by laws of physics	infinitely malleable

- software changes are more frequent and more drastic than changes of hardware

Hardware – software analogy

- life is easy as long as software components behave like hardware components
  - modularization, information hiding, interfaces
- when hard/soft analogy breaks - problems start
  - at times the “lego model” does not fit software
  - change cannot be localized and propagates across components
  - methods must better utilize the “soft” nature of software

Q & A

End of Set #4 More on PL