Product Line Approach

Part I: Reuse, Product Lines, domain analysis

Tutorial at GCSE'200
Erfurt, October 2000

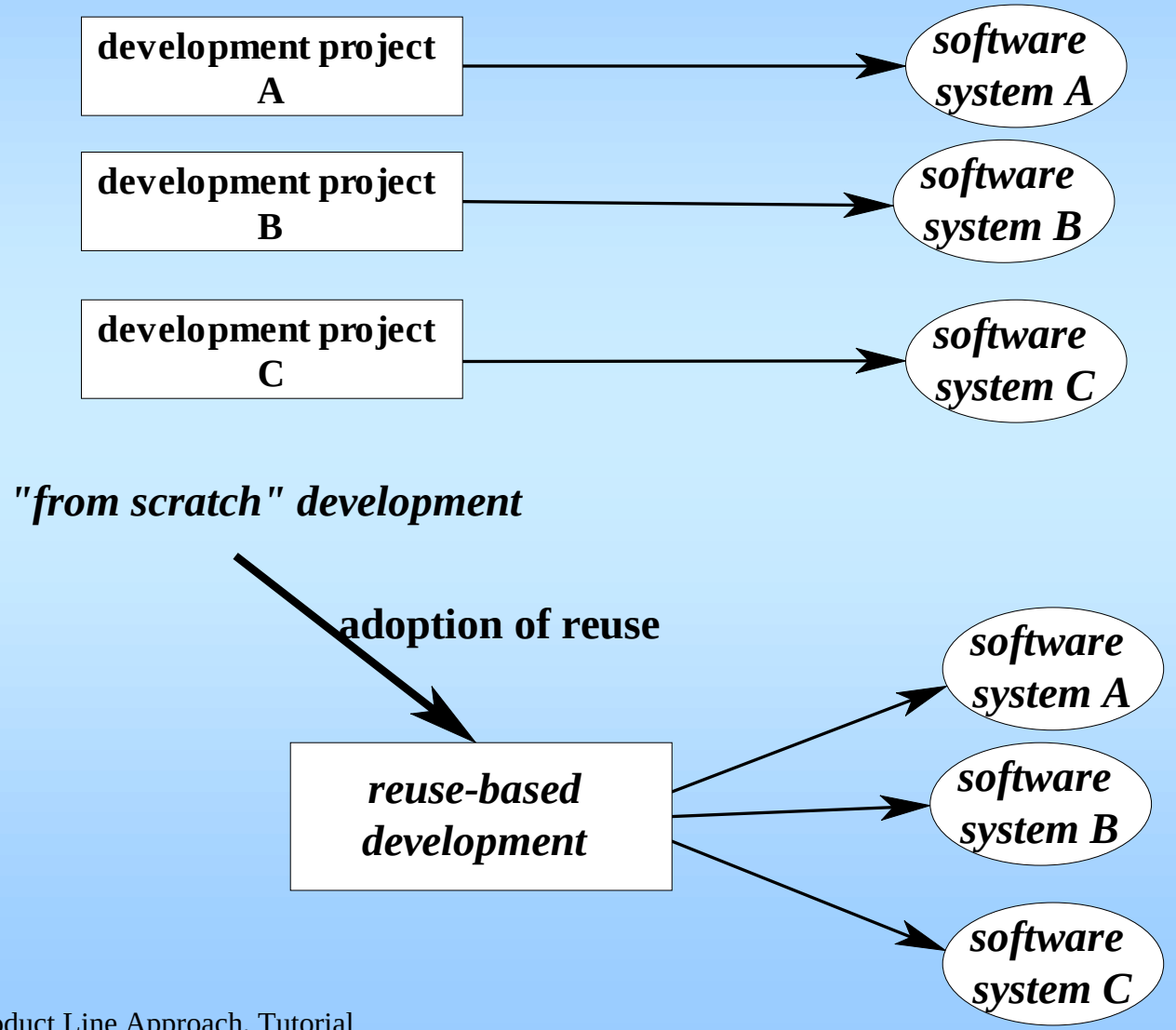
Stan Jarzabek
School of Computing
National University of Singapore

Tutorial overview

- Shift to reuse-based software development
- Systematic support for Product Lines
  - domain engineering
  - reuse-based system engineering
- Domain analysis

- Generic architectures for a Product Line
- Component-based development
- Why do we need a component construction environment?
- Case study: Facility Reservation Systems
- Conclusions

Transition to reuse-based software development



DfR and DwR processes



Domain experts

domain knowledge

existing systems in a domain

Development for reuse *DfR*

Feedback

reusable software assets: architecture, components, etc.



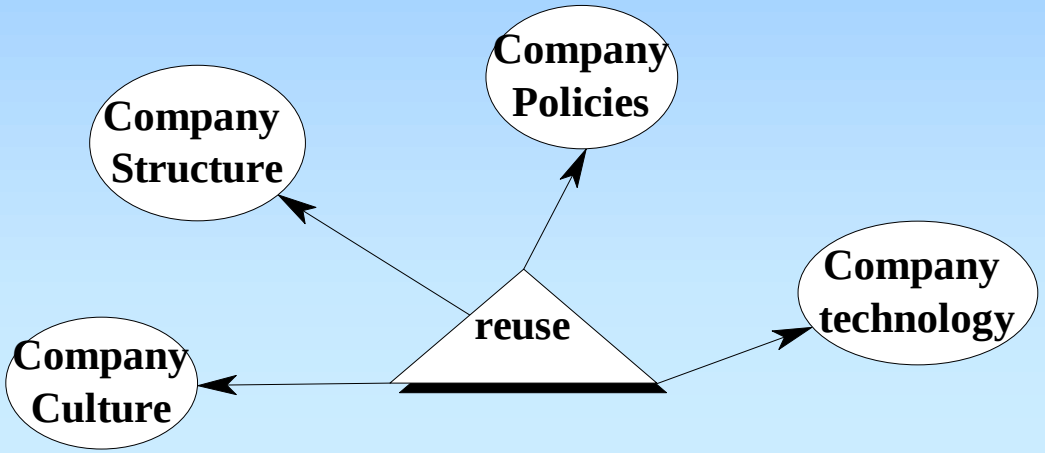
Product development team

user requirements

Development with reuse *DwR*

software product

Reuse affects a company in many ways



- company culture: develop for others and use others work
- company policies: setting up reuse procedures, reward system, monitoring reuse
- company structure: reuse manager, developers for reuse, domain analyst
- company technology: DfR and DwR methodologies

Approaches to reuse

- libraries of functions, classes
- information hiding (Parnas), pre-processing
- OO approach
- software architectures
- application generators
  - compiler-compilers, 4GL, CASE tools, visual environments
- component-based software engineering (CBSE)
- domain-specific, Product Line approach
  - reuse focused on families of similar software systems

Examples of Product Lines

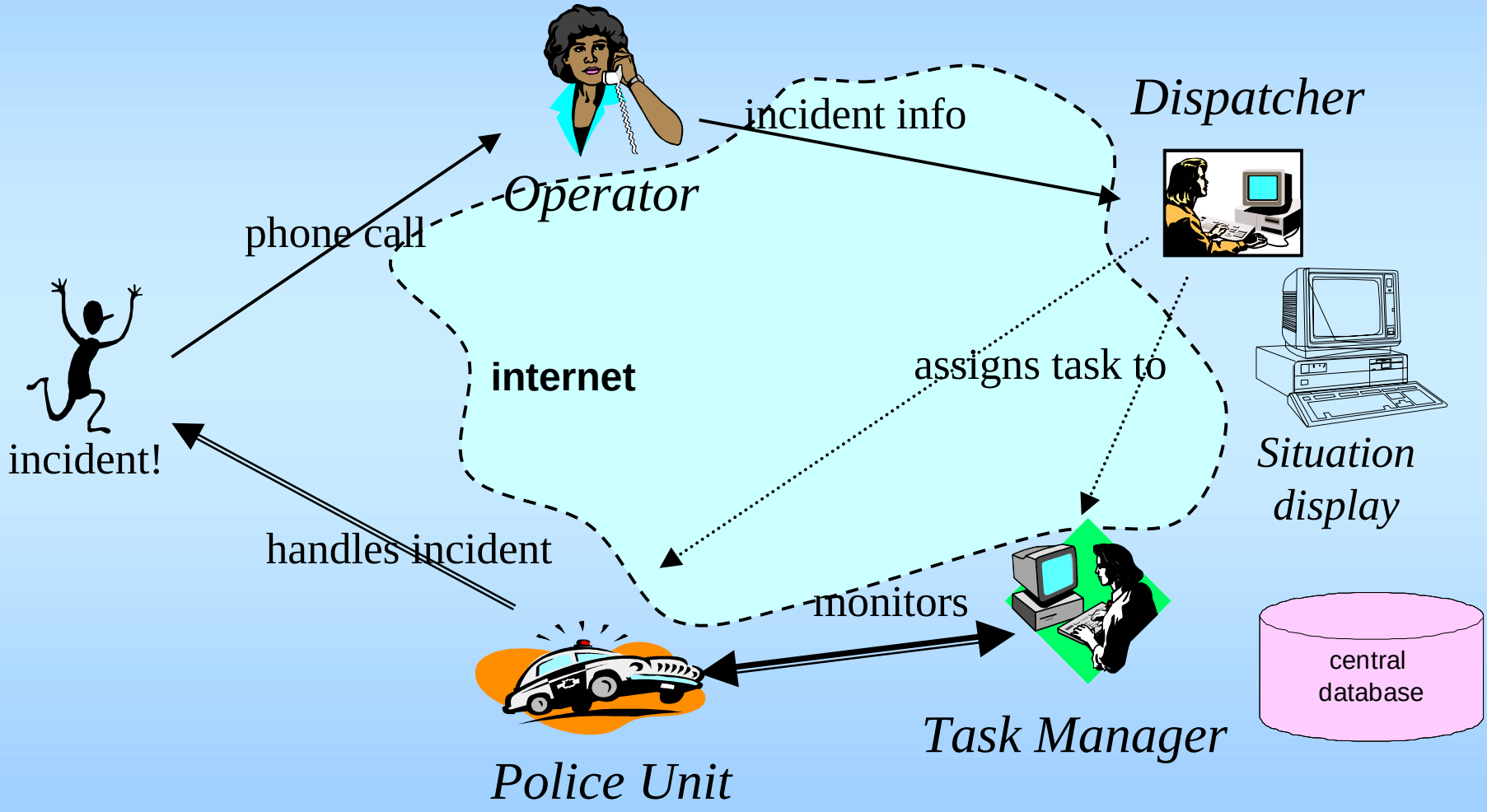
- similar systems that differ in functionality:
  - Enterprise Resource Planning Systems in different companies
  - library systems: for universities, companies, public library
  - Facility Reservations Systems: for companies, hotels, etc.
  - Computer Aided Dispatch Systems: for police, fire & rescue
- different systems that share a common GUI, DBMS
- application suites:
  - MS Office: WORD, Excel, Access
  - ERP: accounting, customer order processing, etc.
- successive versions of the same software product developed over years of changes

Facility Reservation Systems (FRS) Product Line

- FRSes are needed to reserve facilities in:
  - universities, companies, hotels/inns, aeroplanes, etc.
- Requirements for FRS PL:
  - Common requirements:
    - Reservation Management
    - Facility Management
  - Variant requirements:
    - User Management
    - User rights definition
    - Payment

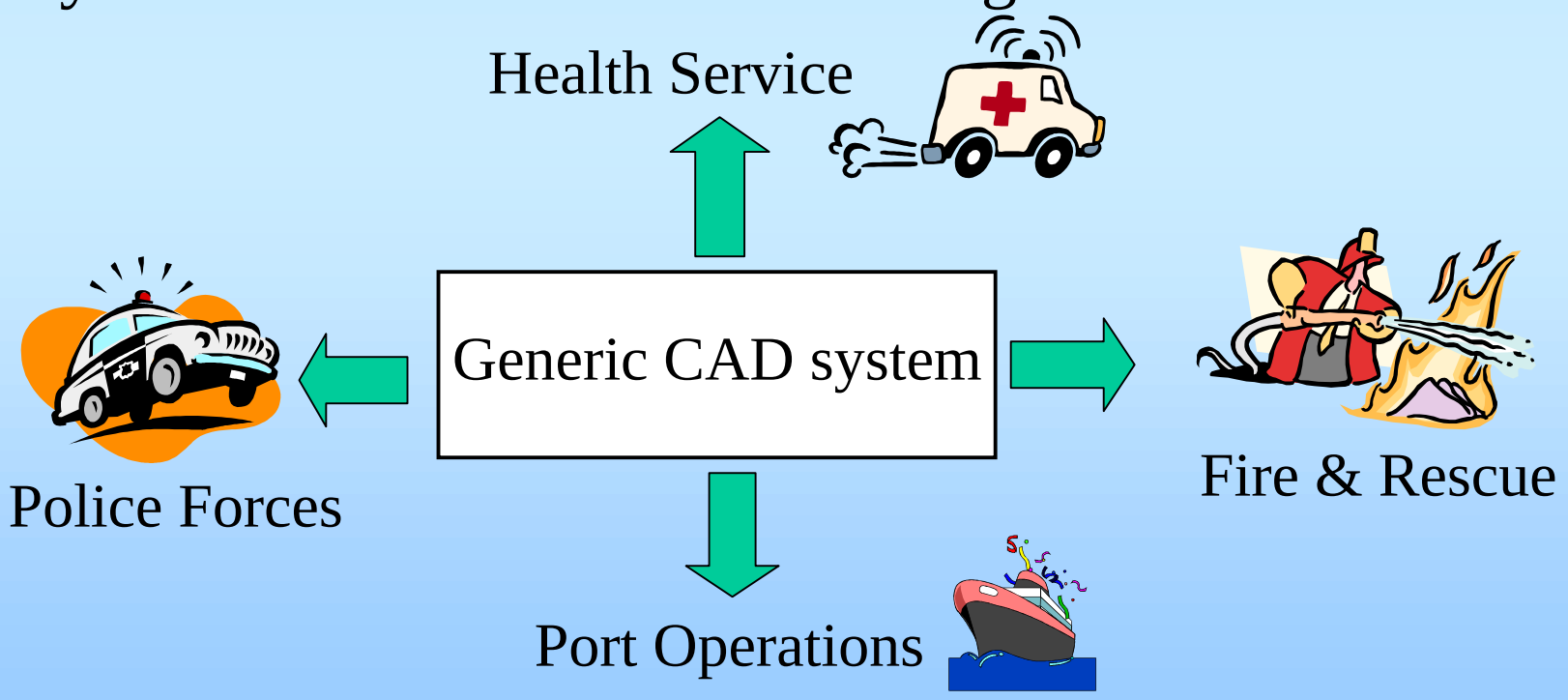


Computer Aided Dispatch (CAD) for Police



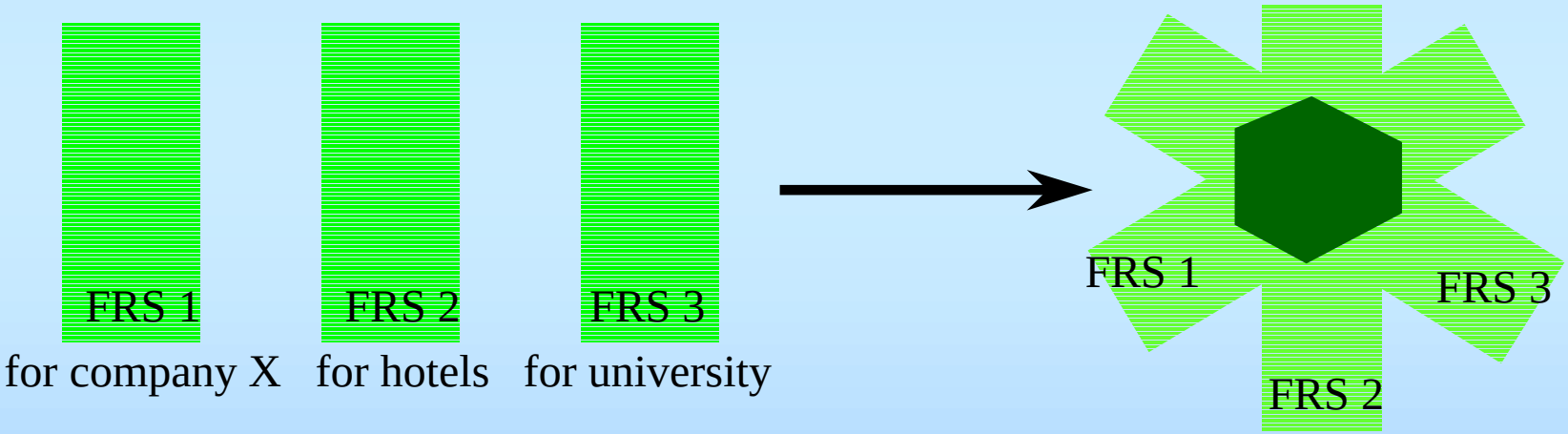
CAD Product Line

- Fire & Rescue, port, etc. display a similar operation pattern
- reuse: development and maintenance of CAD systems from a common base of generic software



A Product Line (PL)

- Product Line (PL) is a family of similar systems:
  - PL members share common requirements
  - PL members differ in certain ways (variants)



- development & maintenance of PL members from a set of reusable assets
- goal: save development and maintenance cost

What does it take to support a PL?

- find sponsors for reuse among top managers
- select and scope a PL
- study existing systems in a PL
- build system prototypes
- understand commonalties and variants across PL members, gather requirements for a PL
- build a generic PL architecture
- manage changes (people, policies, organization)
- adopt methods and tools to build products with reusable assets

Selecting and scoping a Product Line (PL)

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

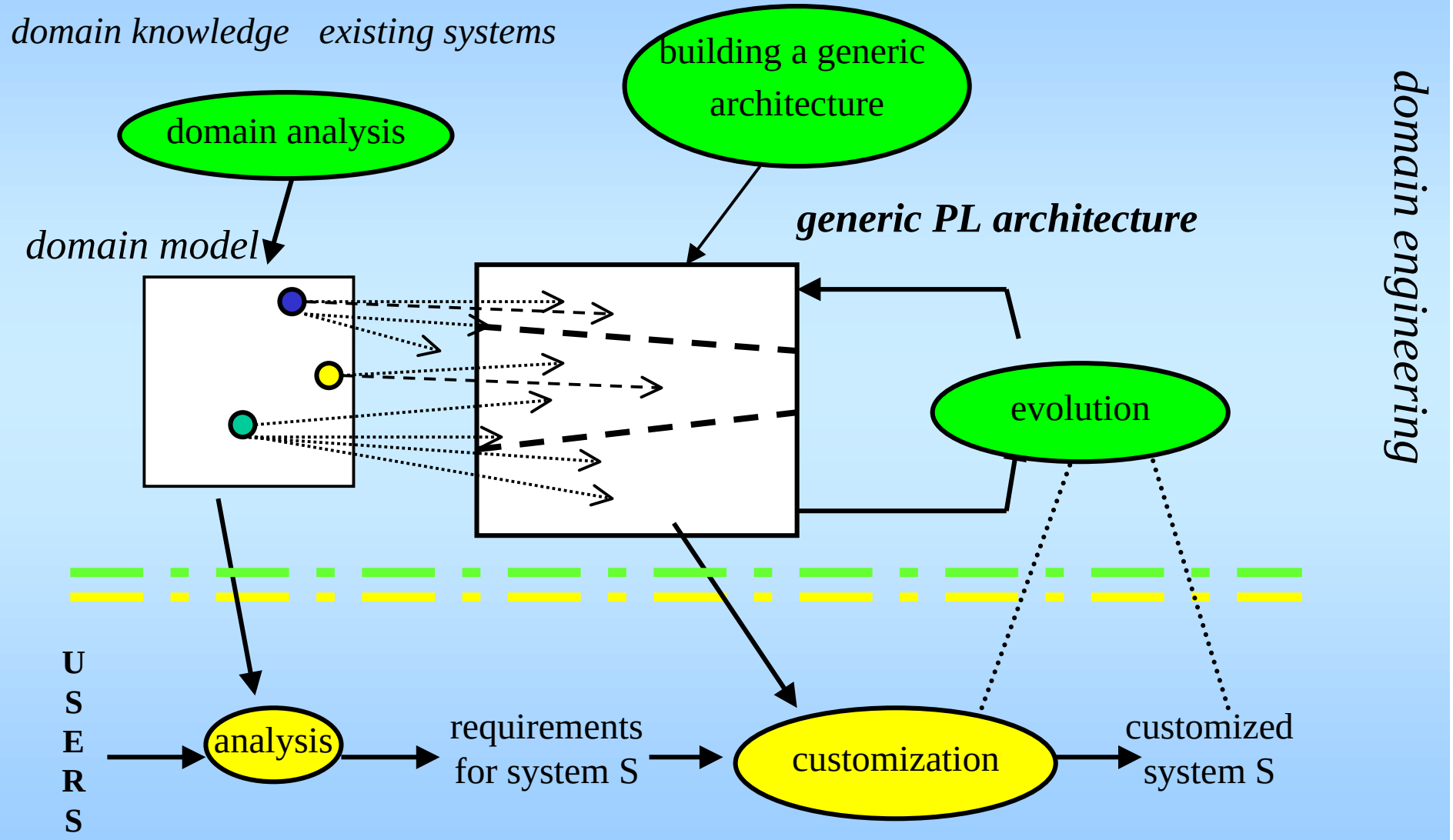
Domain engineering

domain: area of knowledge including knowledge how to build software systems (or part of them) in that area

domain engineering is Development for Reuse (*DfR*):

- identifying, describing and storing reusable assets based on:
  - past experiences in building systems (or parts of them) in a domain
  - knowledge elicited from domain experts
- reusable assets may include any of the following:
  - a domain model: description of common and variant requirements
  - a generic architecture (design and implementation)
  - test cases, documentation, etc.
- providing means (methods and tools) for reusing these assets when building new systems

Domain engineering and system engineering (I)



Technical problems in domain engineering

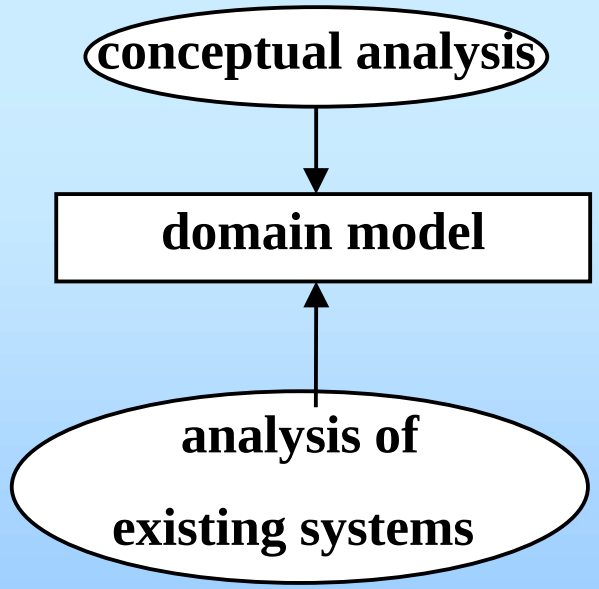
- how to describe common and variant requirements?
  - how to model variants in UML?
- how to design a generic PL architecture?
  - there is a long list of techniques: information hiding, table-driven methods, pre-processing,, OO frameworks, …
  - which technique will be most effective for a task at hand?

Design qualities:

- will it be easy or difficult to reuse a generic architecture during system engineering?
- will it be easy or difficult to evolve a generic architecture as a business environment evolves?

Domain analysis

- modeling common and variant PL requirements
- domain analysis process:
  - top-down: takes into account conceptual similarities
  - complemented by bottom-up analysis of existing systems



goals of domain analysis:

– to gather requirements for a generic PL architecture

– to explain a generic PL architecture to system engineers

Domain analysis vs. requirements analysis

*domain analysis is like system requirements analysis **but***:

- addresses a family of systems rather than a single system
  - so we need to model common and variant requirements
- requirements can appear in different configurations in different systems, PL members
  - so we need to describe legal configurations of requirements
- PL requirements may refer to:
  - functional system properties
  - non-functional system properties
  - architecture (e.g., client-server, component-based)
  - design decisions (e.g., the choice of sorting algorithm)
  - implementation, platform (e.g., in C++, DOS, using Access DBMS)

How to analyze a domain?

- build specific systems to learn about a domain and see problems
- describe a typical system (*domain defaults)*
  - e.g., using UML diagrams
- model variants and variant constraints
  - e.g., using FODA methods
- describe variants in respect to domain defaults
  - e.g., using extended UML diagrams

Facility Reservation Systems (FRS) Product Line

- FRSes are needed to reserve facilities in:
  - Academic institutions, hotels/inns, aeroplanes, offices, etc.
- Requirements for FRS PL:
  - Common requirements:
    - Reservation Management
    - Facility Management
  - Variant requirements:
    - User Management
    - User rights definition
    - Payment



Common and variant requirements for FRS domain

An FRS helps company staff manage (add, delete, modify) reservation of facilities. Facilities are rooms and equipment (such as PCes or OHPs).

± Different companies such as universities, hotels, hospitals and companies, have different physical facilities and rules for their reservation.

± FRS will allow one to view reservations by facility and/or by date

± Sometimes, an FRS should also manage users, facility types and physical facilities.

± Payment may be required for reservation and cancellation in certain companies. Printing bills may be required.

FRS domain defaults

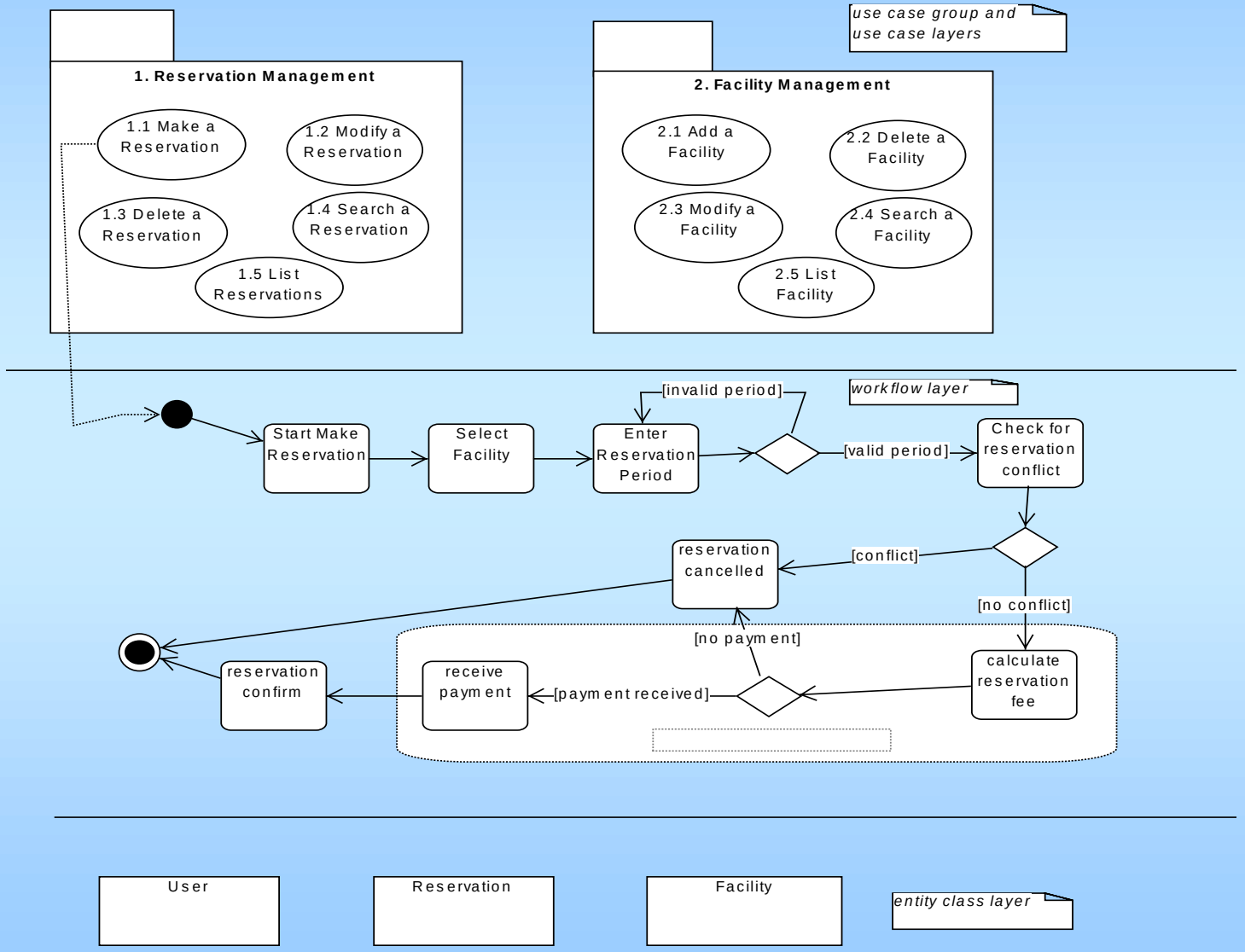
Reservation Management:

- Make a Reservation
- Delete a Reservation
- Modify a Reservation
- Search/Retrieve Reservations

Facility Management:

- Add a Facility
- Delete a Facility
- Modify a Facility
- Search/Retrieve Facilities

Modeling domain defaults



Default use case *Make Reservation*

If "USER" has a preference

    "USER" searches through "FACILITY"

IF "USER" does not have a preference

    System searches through "FACILITY

System selects "FACILITY" to reserve on behalf of user

"USER" supplies time period of reservations

If 'FACILITY" is reserved at the specified time

    "USER" repeats the reservation use case

// *Special Requirements*:

System should respond to use input within "RESPONSETIME"

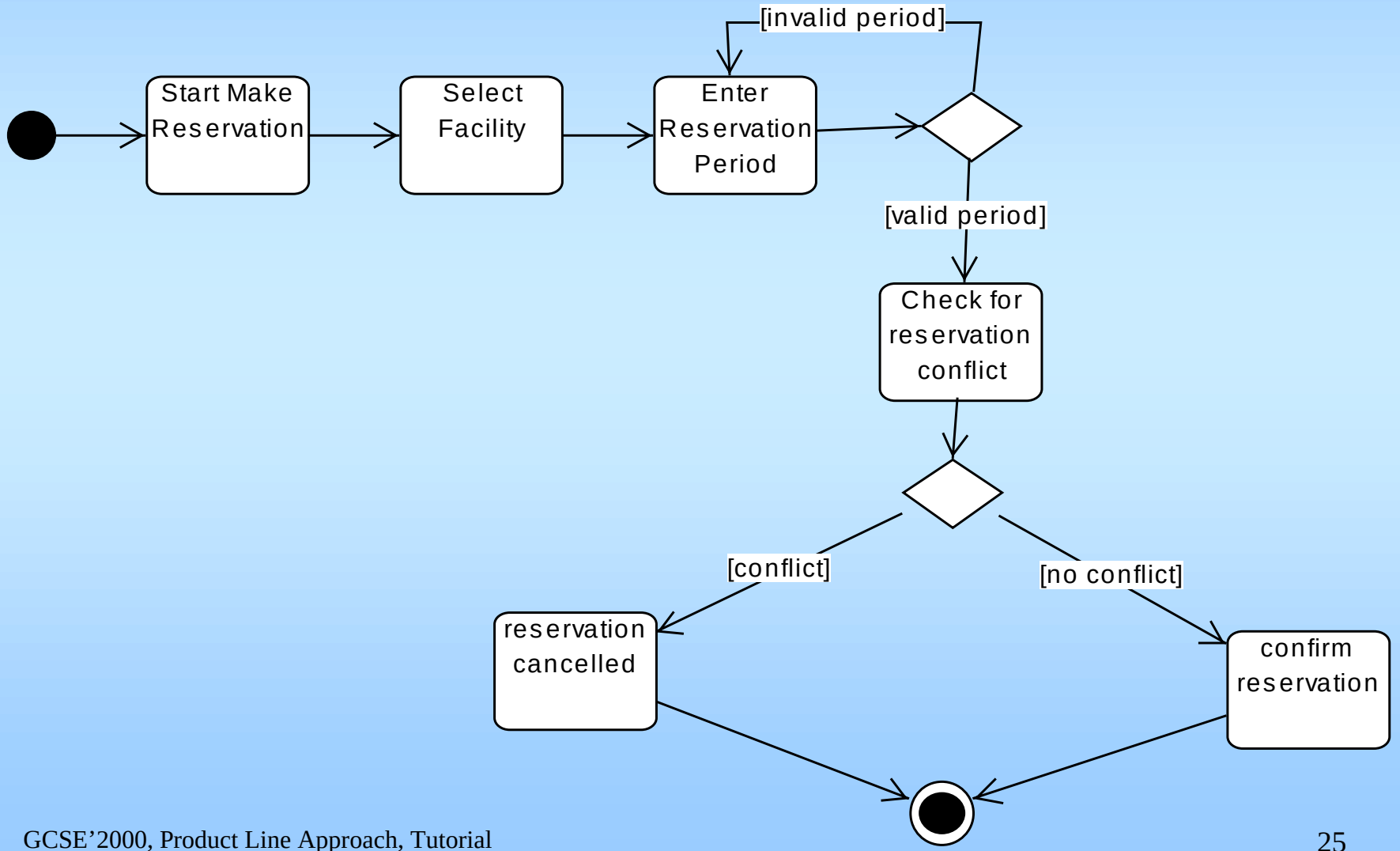
// The default values of the variables are listed below:

USER = user

FACILITY = facility

RESPONSETIME = 30secs

Default workflow *Make Reservation*



Mandatory and variant PL requirements

- each requirement is given a unique name:

  Add_Reservation // capability to add new reservations

- mandatory requirements are common to all the systems in a PL
- variant requirements appear in some of the systems
- variant requirements are classified as:

optional requirements:

Payment-OPT // capability to handle payment for reservations

Bill-Construction-OPT // capability to create bills for reservations

Or and alternative requirements

variant requirements, continued:

OR-requirements – any number of many options:

Retrieve_Reservations_by : Date-OR

Retrieve_Reservations_by : Facility-OR

alternative requirements – only one of many options:

User_Interface : GUI-ALT // graphical user interface

User_Interface : Line-ALT // command line user interface

More examples of optional requirements

Cancellation_Fee–OPT // the system shall charge a certain amount whenever a reservation is cancelled within “n” days of the reservation

Conflict_Resolution–OPT // a conflict arises if a new reservation would result in a clash with other reservation(s); the system will perform some form of conflict resolution

Suggest_Other_Facility–OPT // when a conflict arises, the system shall suggest other similar facilities that are available for the time period required

Retrieve_Available_Facilities–OPT // given a particular time period, the system shall produce a list of facilities available during this time period.

Dependent requirements: payment

- PL requirements can appear in different configurations in any given system, PL member
- only certain configurations of requirements are legal

For example, in any given FRS, any of the following requirements can hold only if the requirement of payment holds:

- reservation cost, computation of payment
- discounts
- printing bills
- cancellation fee

We say that above requirements depend on payment

Dependent requirements: notation and examples

R1 depends on R2 if in any given system, requirement

R_1 can hold only if R_2 also holds

notation: R_1 R_2

Bill_Construction–OPT Payment–OPT

Cancellation_Fee–OPT Payment–OPT

Suggest_Other_Facility–OPT Conflict_Resolution–OPT &

Retrieve_Available_Facilities–OPT

Notation for dependent requirements

In any given FRS:

R_1 **R_2** R_1 can hold only if R_2 holds

R_1 **! R_2** R_1 can hold only if R_2 does not hold

R_1 **< R_2 | R_3 | … | R_N >** R_1 can hold only if any (i.e. at least one) of the requirements R_2, R_3, …, R_N holds

R_1 **< R_2 & R_3 & … & R_N >** R_1 can hold only if all of the requirements R_2, R_3, …, R_N hold

[R1, R2, …, R_N] either all of the R_1, R_2, …, R_N hold or none of them holds

FODA: feature-oriented domain analysis

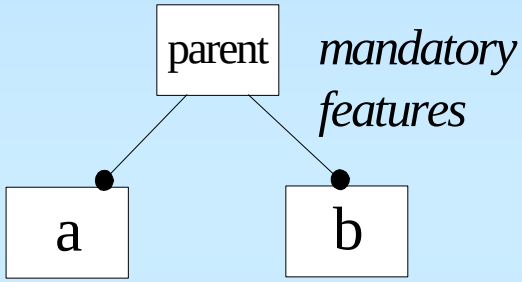
FODA was developed at the Software Engineering Institute

- in FODA, PL requirements are called features
- *feature*:
  - a prominent, user-visible aspect, quality or characteristics of a system
  - any system characteristic (functional + any other requirements)
- FODA: information model, feature model, operational model

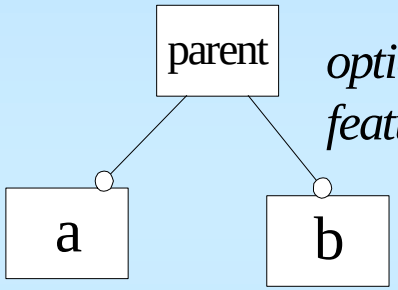
we shall use terms *feature* and *requirement* interchangeably, using term feature mainly when referring to FODA

Feature diagrams

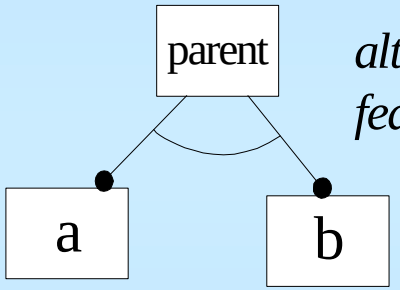
diagrams depict mandatory and variant features as a tree-like hierarchy (*consists-of* relationship among features):



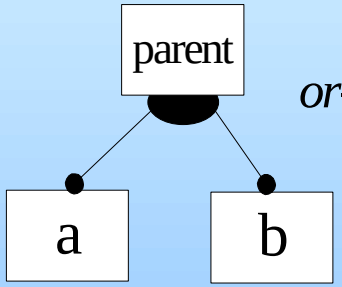
mandatory features



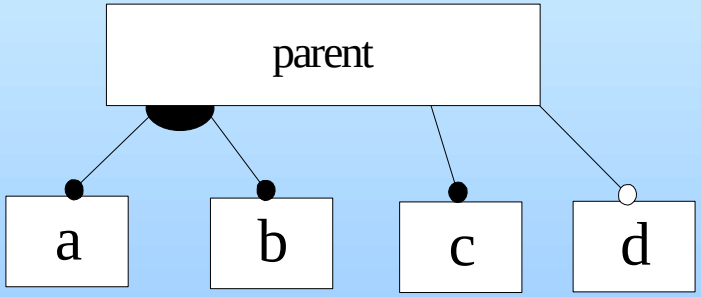
optional features



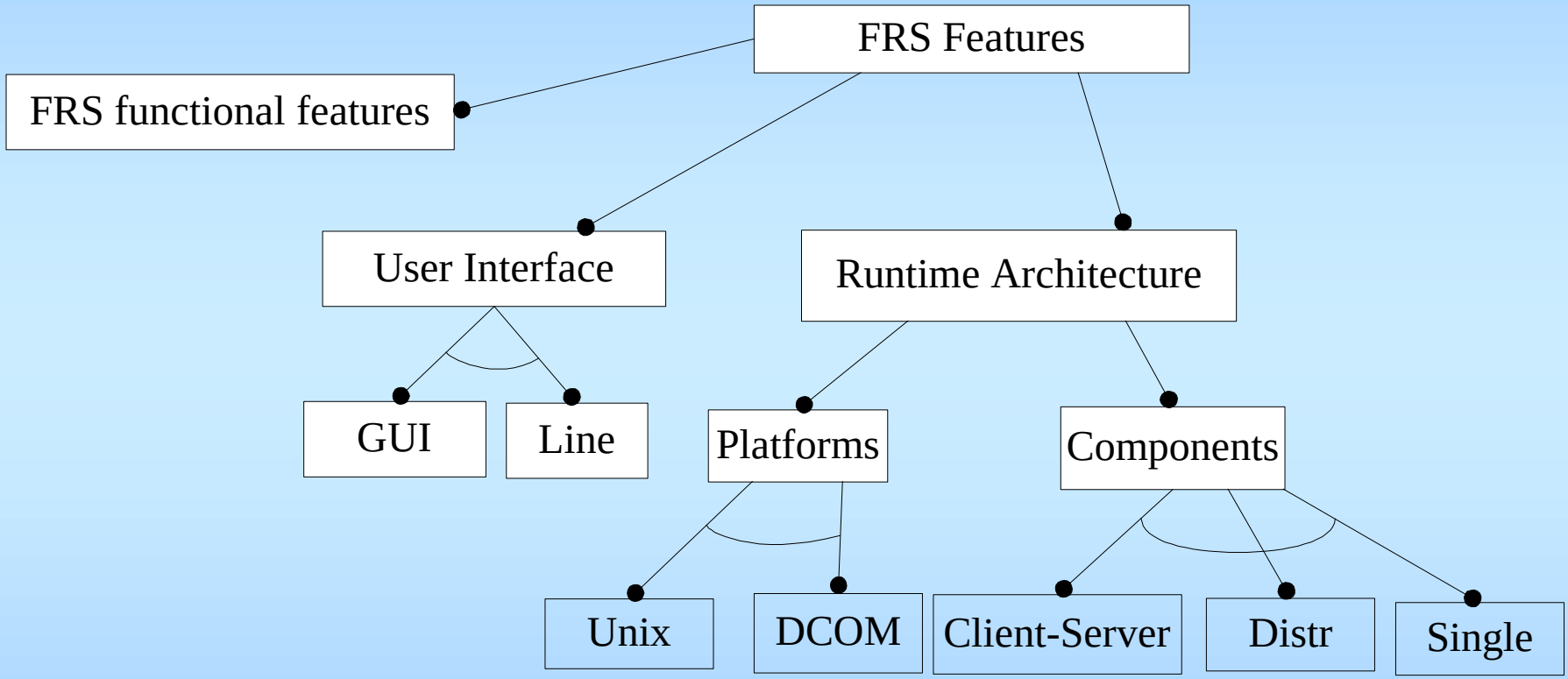
alternative features



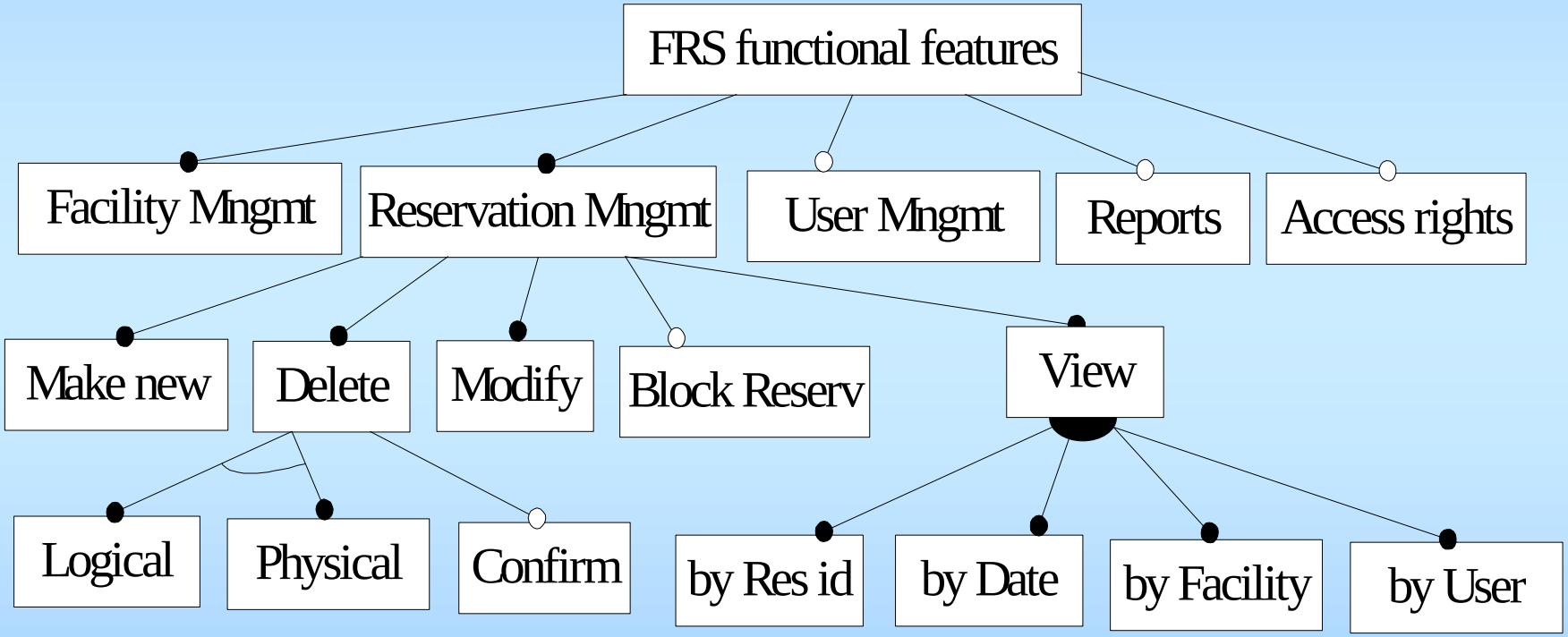
or-features



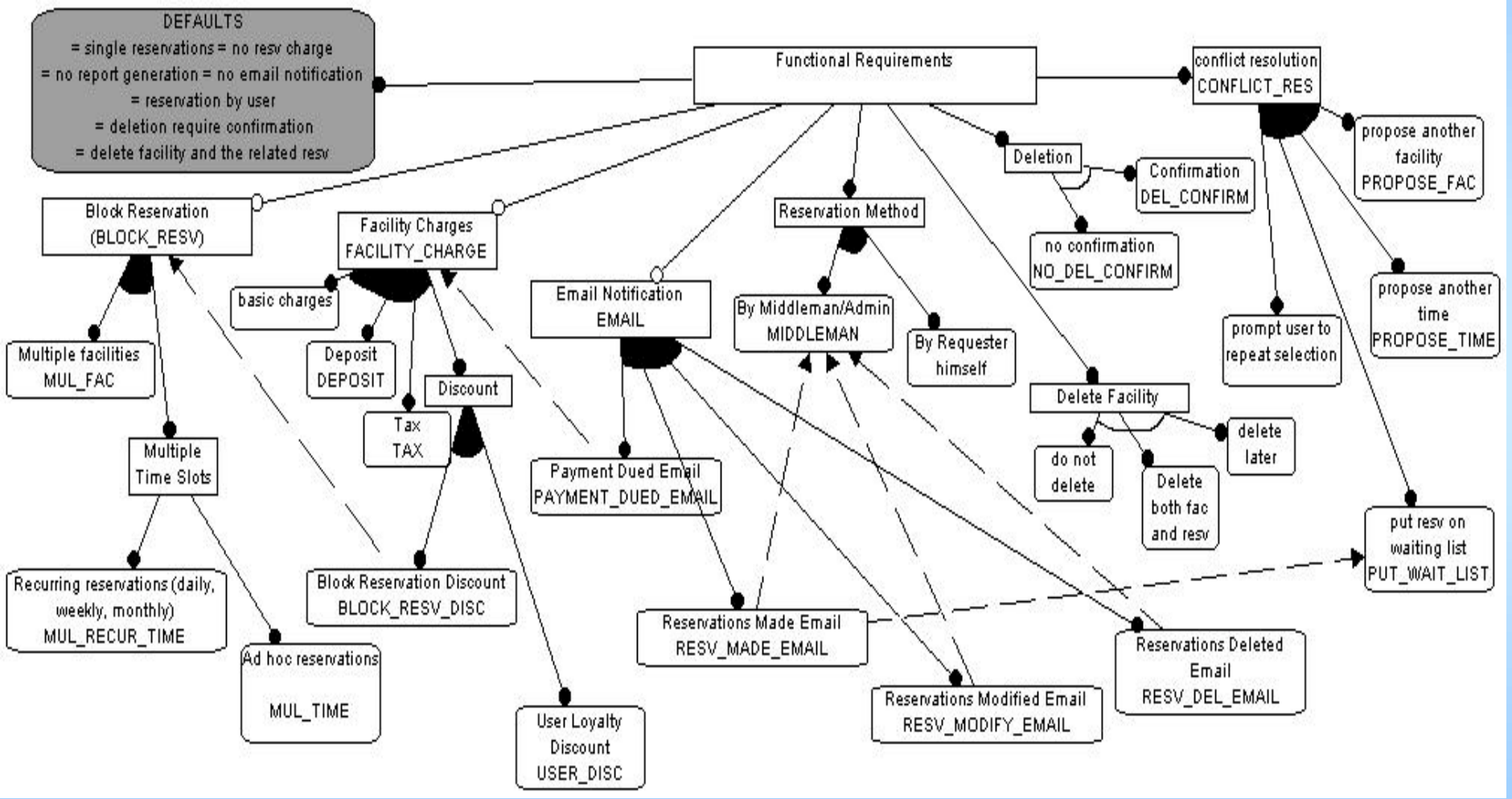
Features in the FRS domain



Functional feature diagrams in FRS domain



Feature diagrams with dependencies



Modeling variants in respect to domain defaults

- use cases and scenarios
- information model (Entity-Relationship or class diagrams)
- activity diagrams
- State Transition diagrams
- Data Flow diagrams
- and many others

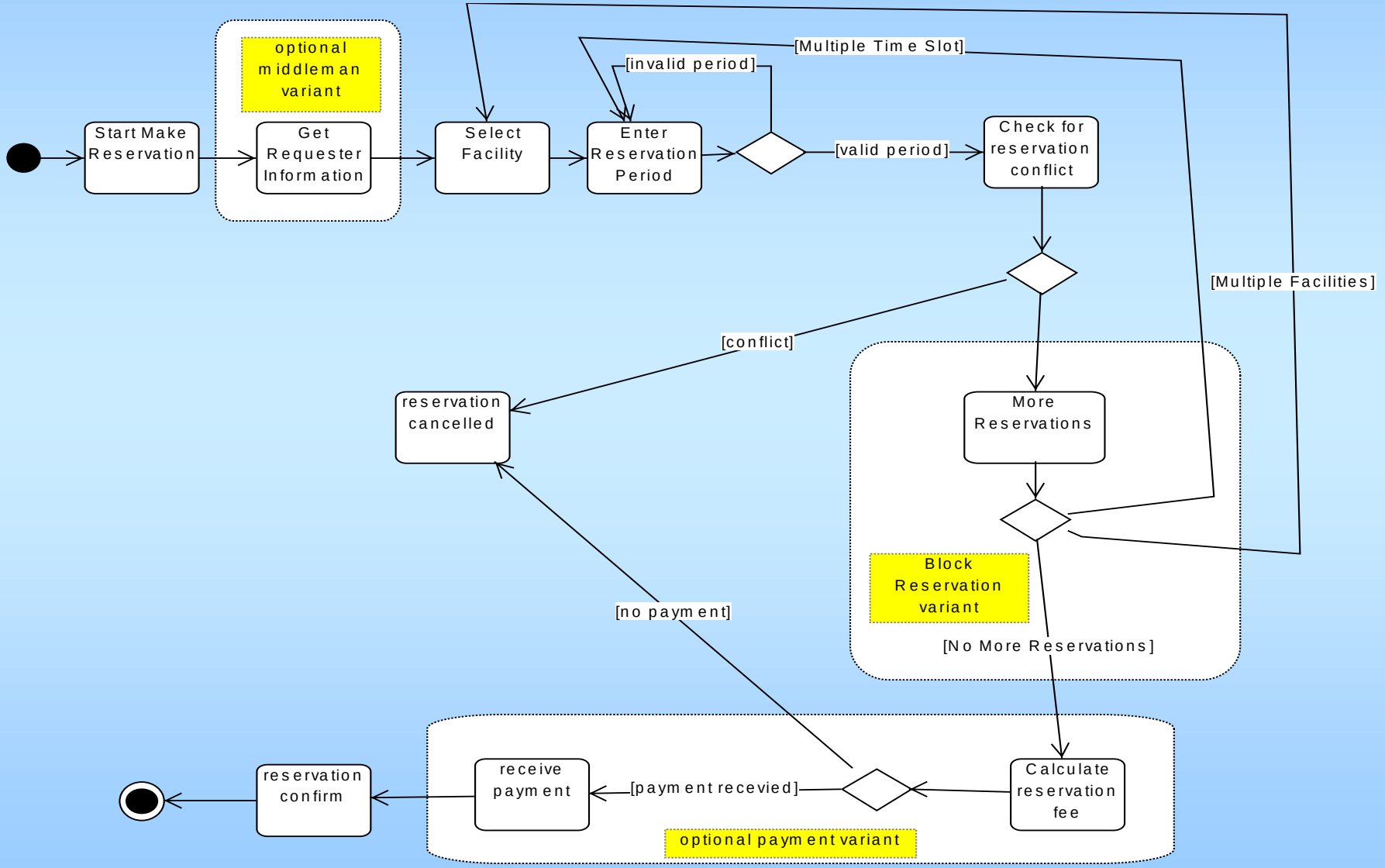
as these notations are meant for modeling one-of-the-kind system; we have to modify the notations to model variants

Use cases with variants

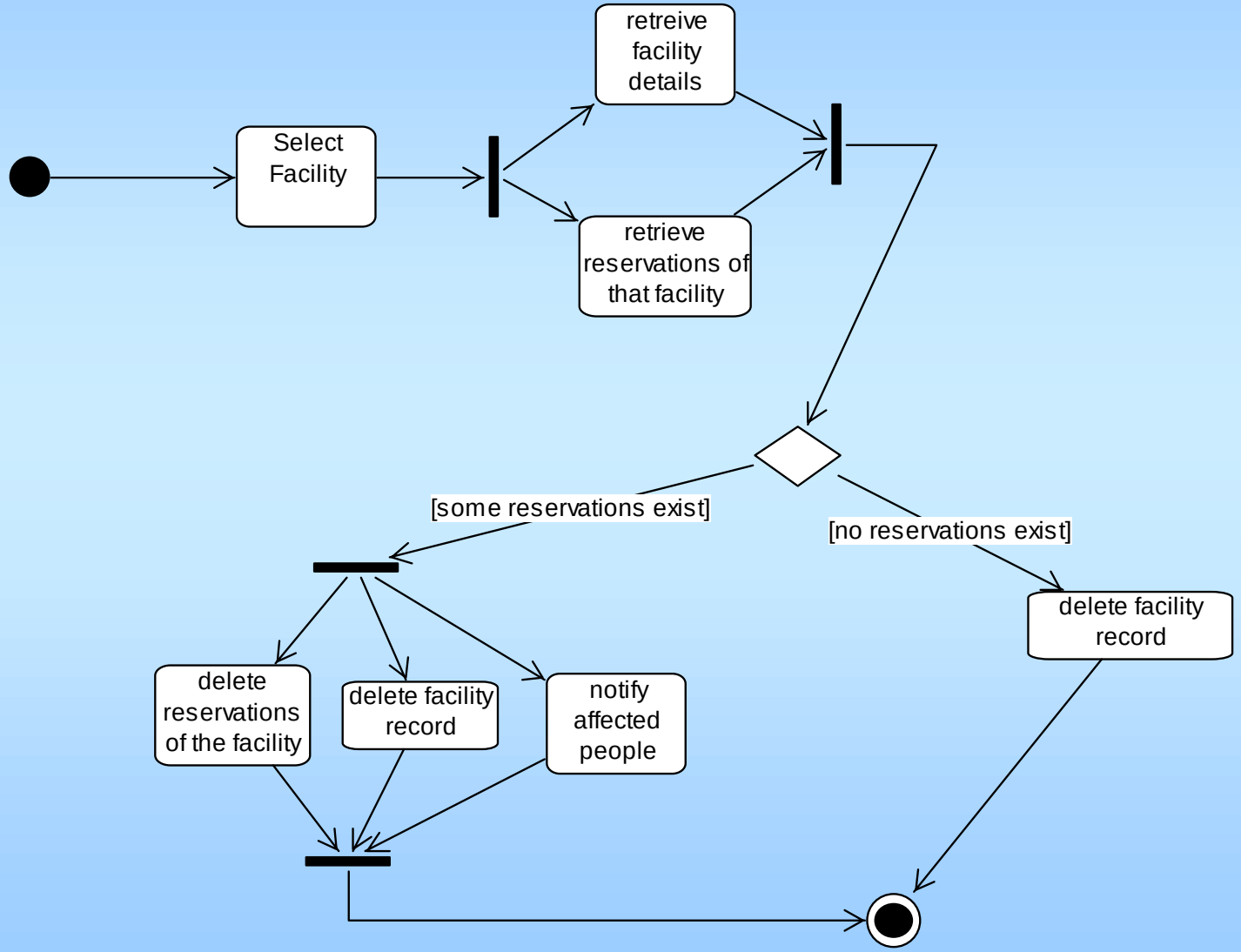
Use case for Making Reservation

1. (No_Pref_Facility-OR): User has no preference for a facility:
   1.1. User supplies time period of the reservation.
   1.2. (View_Available_Facilities–OPT): System supplies a list of available facilities
   1.3. User indicates a particular facility (from this list).
   1.4. (Block_Reservation–OPT): User repeats step 1.3 (at least once) for each additional facility to be reserved.
2. (Pref_Facility–OR): User has no preference for a facility:
   2.1. User specifies the facility to be reserved.
   2.2. System verifies that the facility specified actually exists.
   2.3. (Block_Reservation–OPT):...

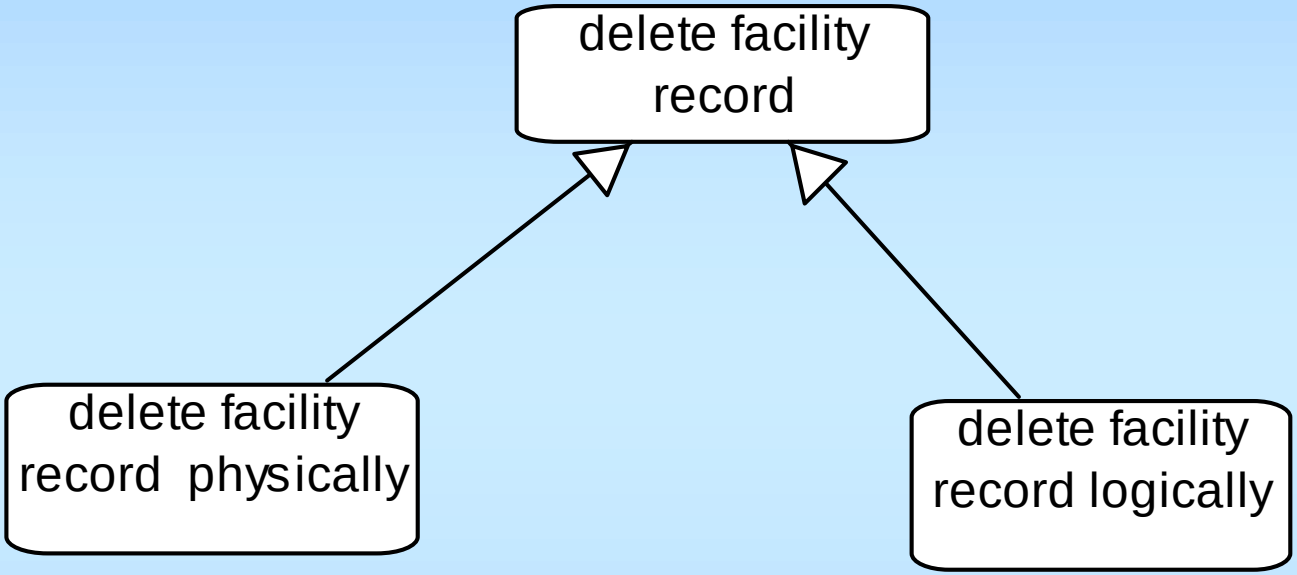
Make Reservation worfklow with variants



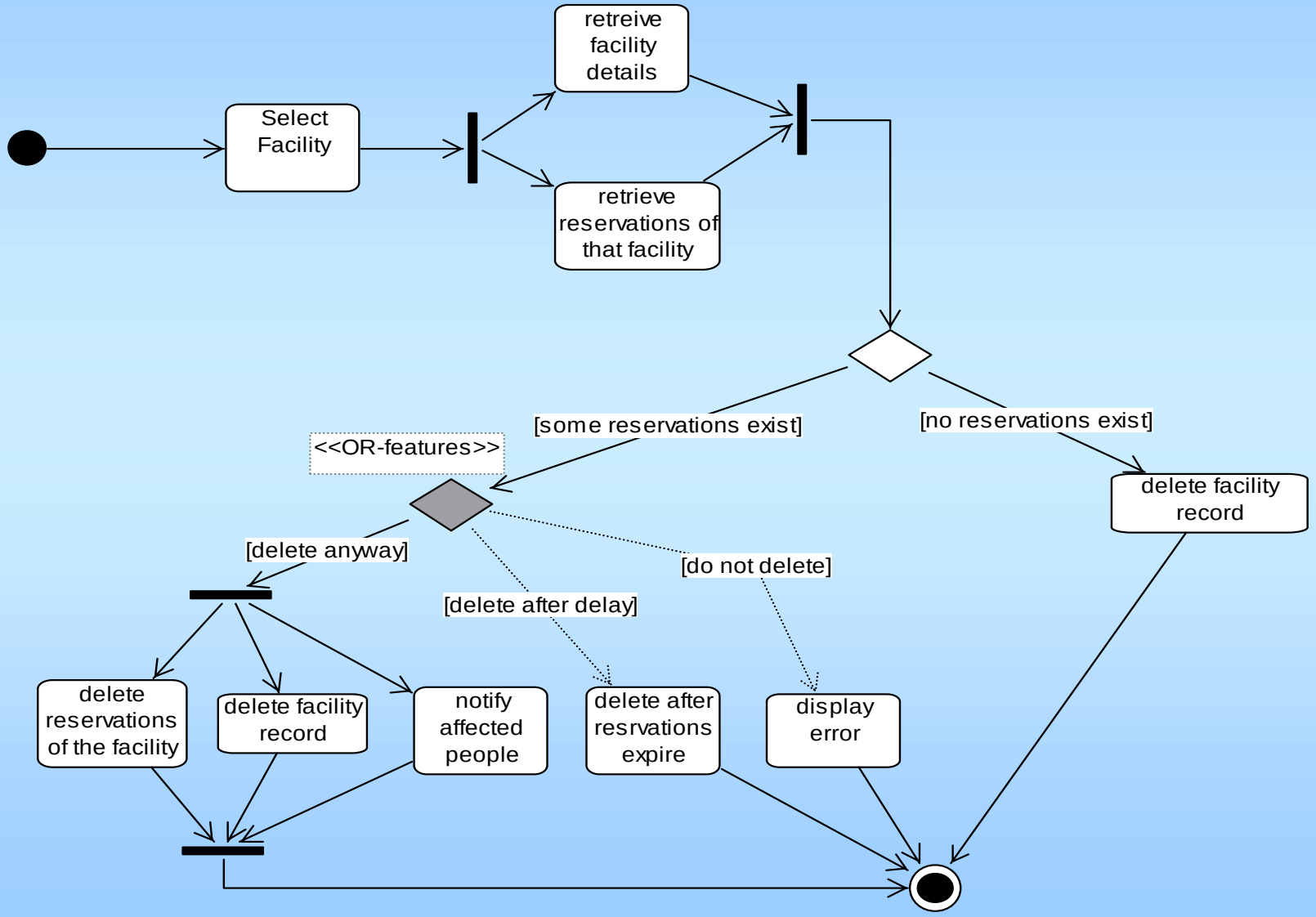
Delete Facility default workflow



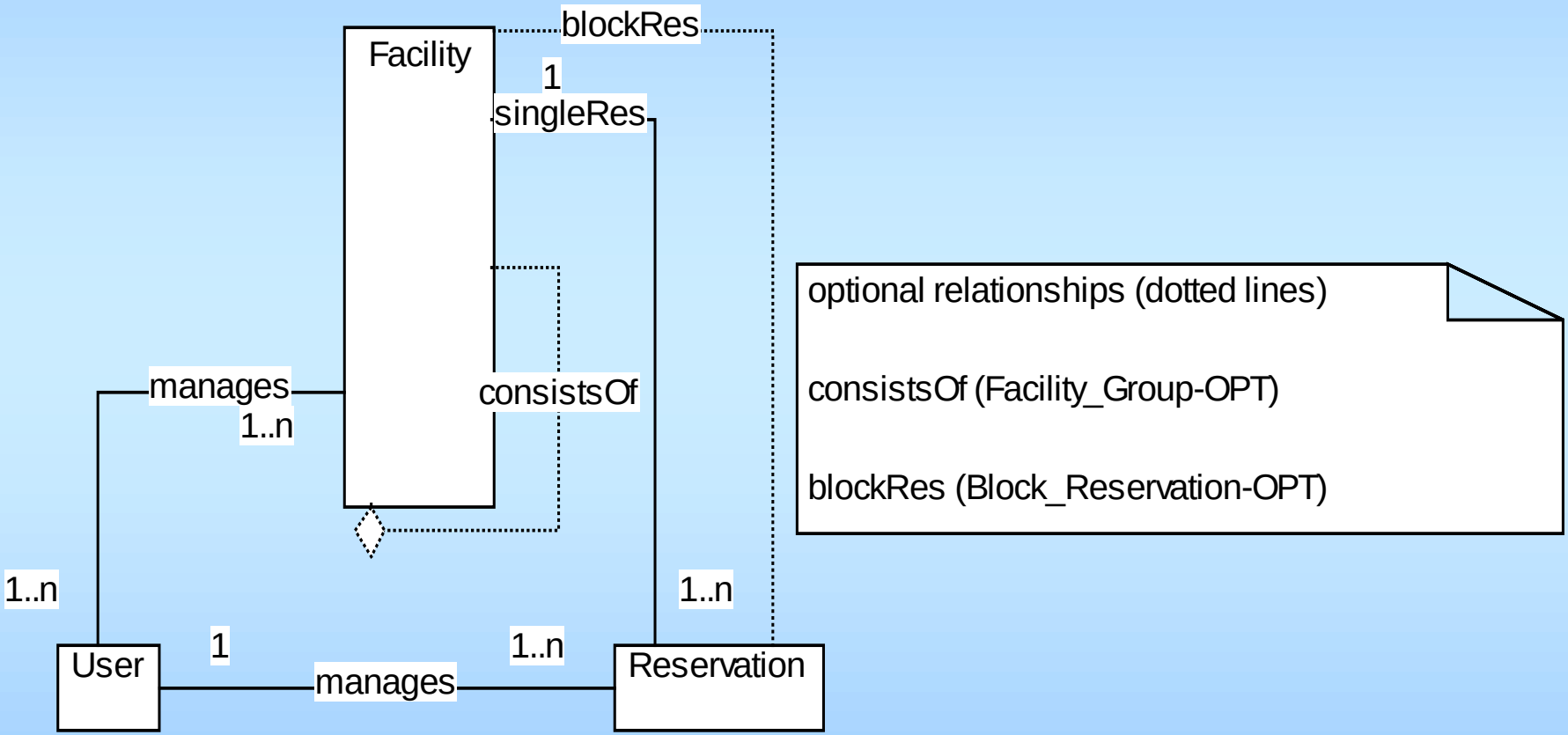
delete facility record variants



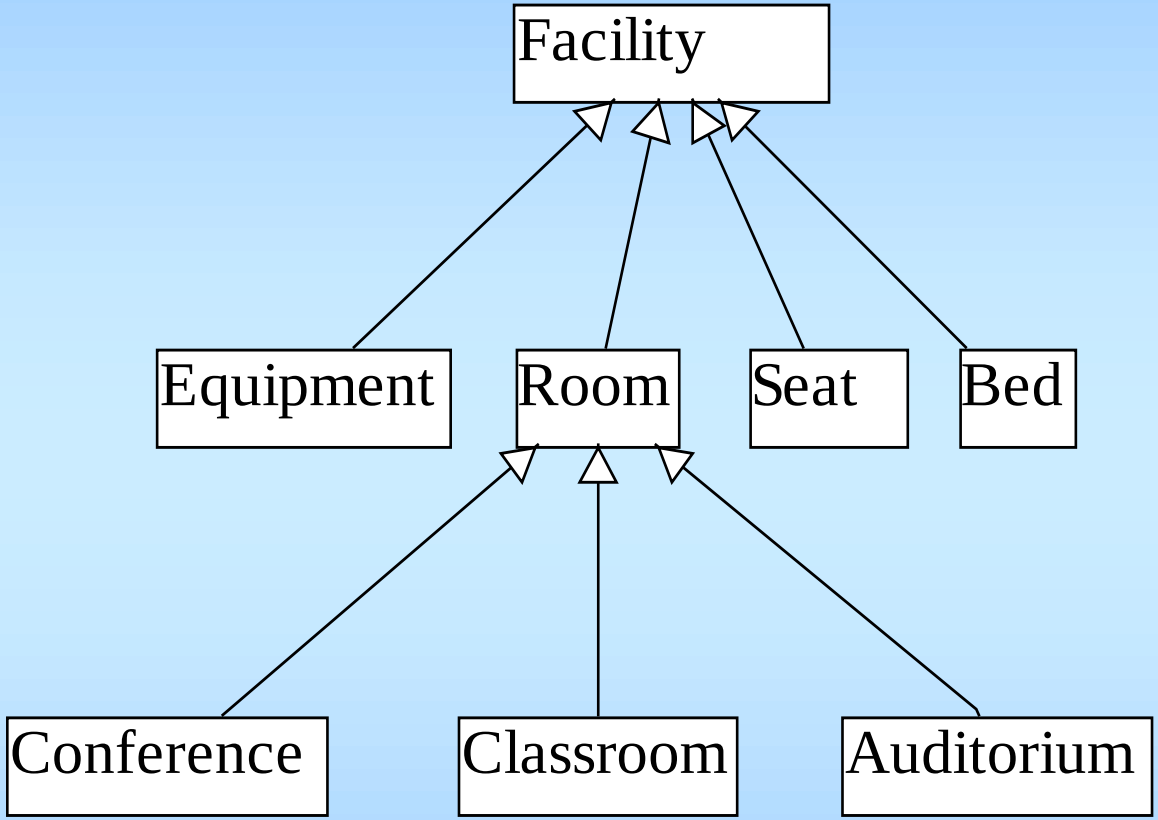
Delete Facility workflow with variants



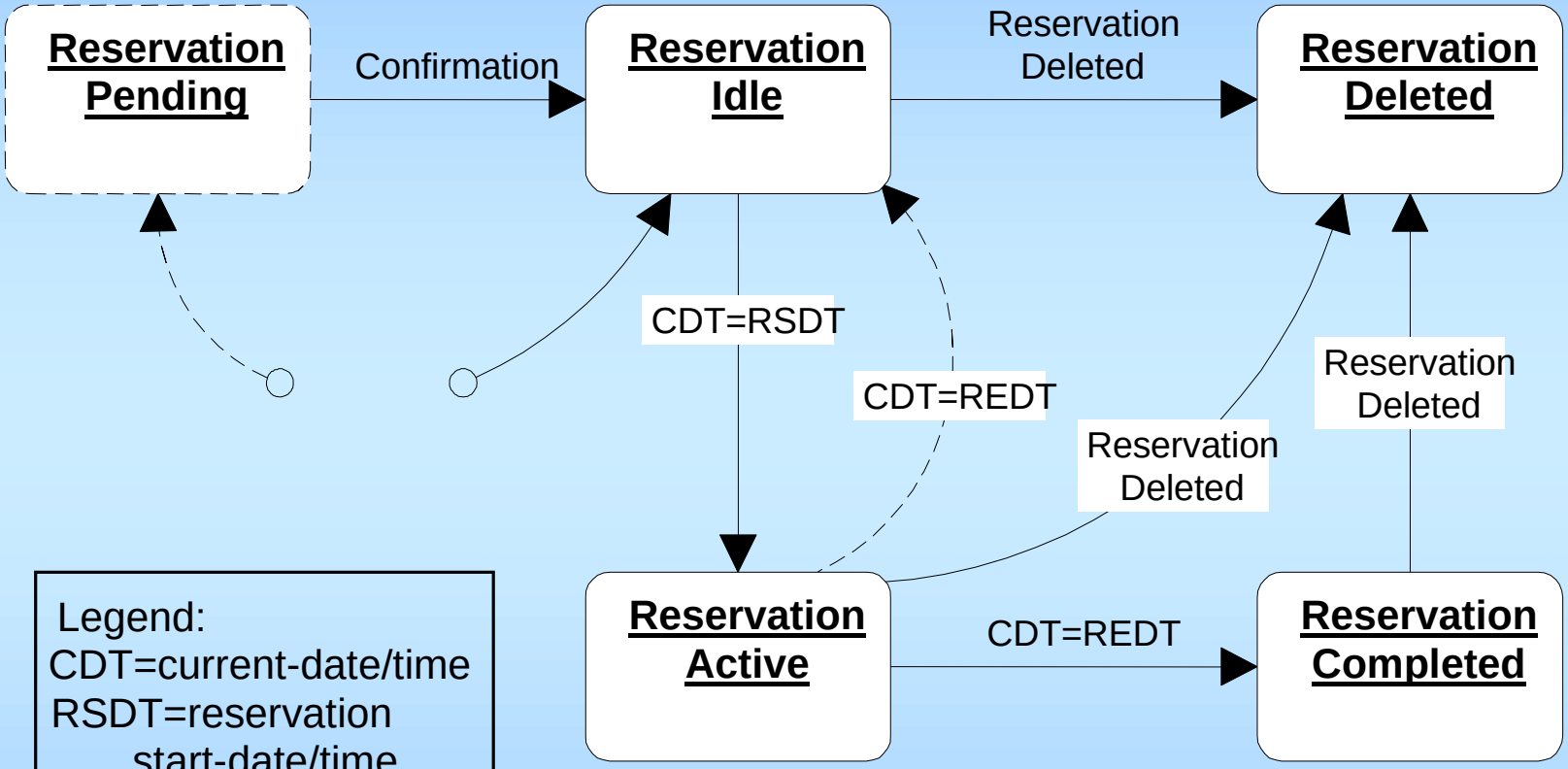
Class diagram



Taxonomy of facility types



State Transition Diagrams



Reservation Pending (Confirm_by_User–OPT)
Confirmation (Confirm_by_User–OPT)
CDT=REDT (Recurring_Reservation–OPT)

Runtime variants: FRS user rights

Who can reserve which facilities and for whom?

R1. all requesters can reserve all facilities but only for themselves

R2. certain requesters can reserve only certain types of facilities

R3. a middleman (admin) can reserve all facilities for all requesters

R4. certain admin can reserve only certain types of facilities for certain requesters

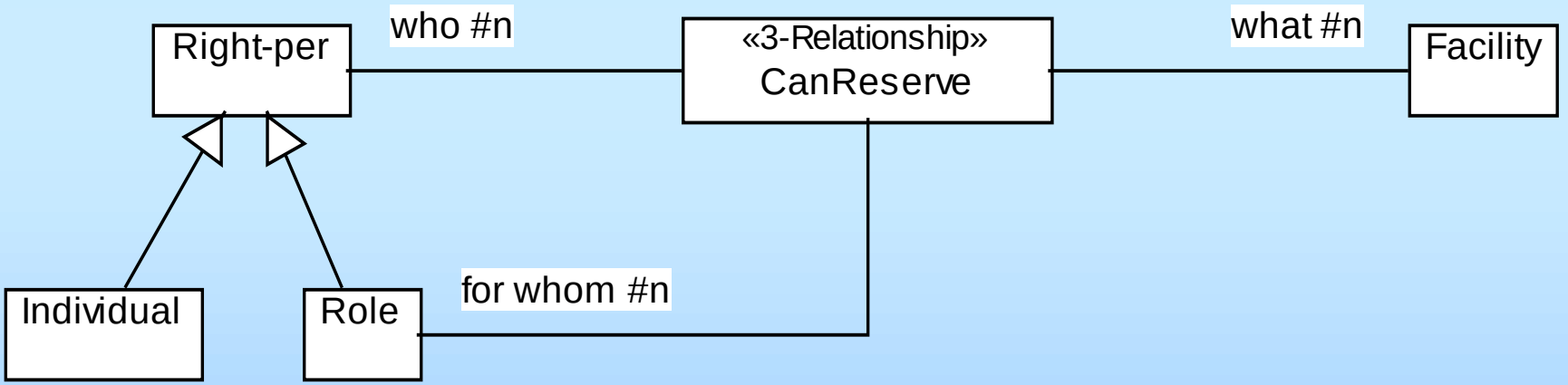
R5. admin can decide who can make reservations of what for whom

So far we assumed that we can enumerate variants. But in user rights the possibilities are endless, it is not possible to enumerate all of them. In addition, user rights are subject to frequent changes and we should be able to change them on the fly, during the FRS session (I.e., user rights should be runtime variants).

A model to capture variants in user rights

User rights are defined as relationship CanReserve (who, what, for-whom) between:

- individual or role (who can reserve); rights can be defined per user or per role (such as Admin, TA, Faculty)
- facility (what)
- role (for whom reservation can be made)



CanReserve (who, what, forWhom)

User right definition

We define user rights for any given FRS as triples, for example:

CanReserve (admin, all, all)

CanReserve (individual, all, self)

CanReserve (Martin Schmid, lecture hall, professor)

- In this solution, instead of enumerating variants, we provide a mechanism to define user rights for any given FRS

In a generic architecture for FRS Product Line, we could design a component called user right control mechanism that will allow one to define user rights and to enforce user rights during FRS session

Decomposing domains to sub-domains

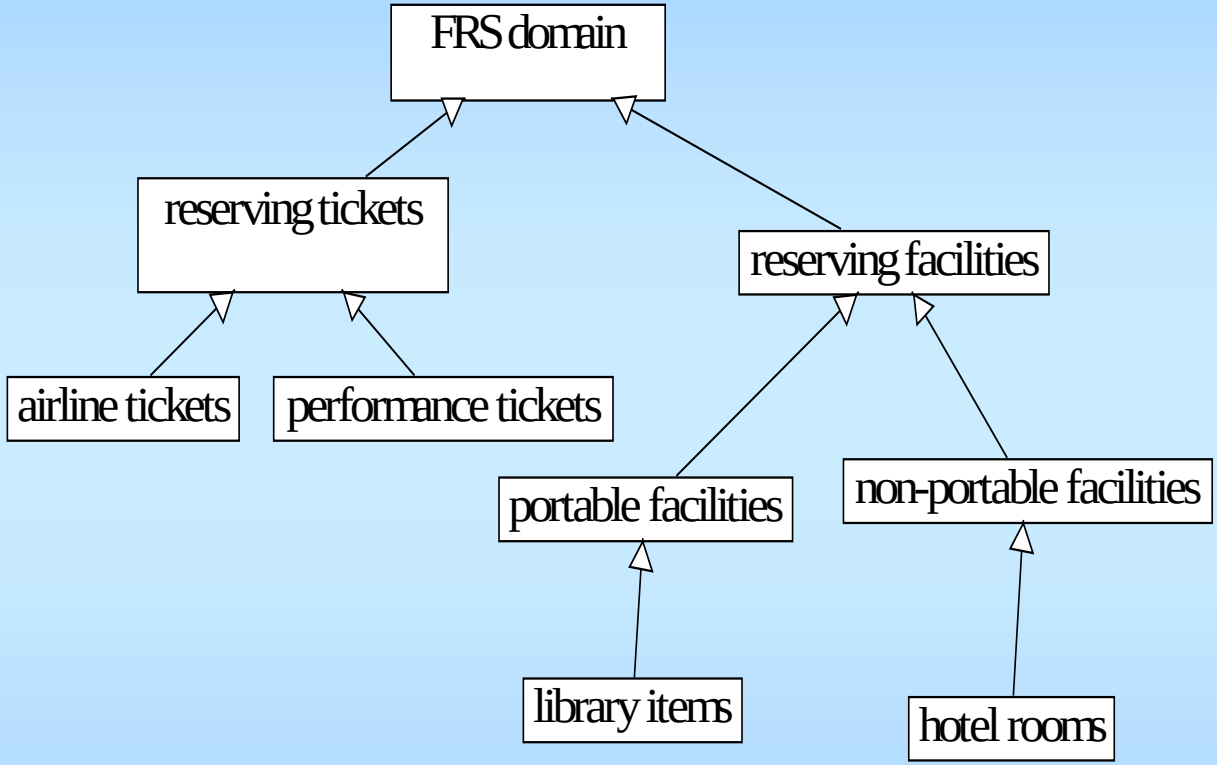
examples of sub-domains

- GUI, DBMS, component communication
- problem sub-domains:
  - user right control, failure detection
  - see book Analysis Patterns by Fowler
- sub-domains can be separately designed as isolated system components
- *IS-A* and *Part-Of* relationship among sub-domains

FRS sub-domains

FRS sub-domain	*description*
academic FRS	facilities: lecture halls, tutorial rooms, OHPs, PCs reservations are done via admin queries: display rooms of given size available on given dates
hotel FRS	facilities: rooms payment for reservation cancellation policies
theater ticket FRS	facilities: tickets
airline FRS	planes for flights tickets for flights
office FRS	many types of facilities, concepts specific to company needs

A taxonomy of FRS sub-domains



Desirable qualities of a domain model:

- Easy to understand
- Sufficiently expressive: domain model should capture various types of user requirements: functional requirements, behavior, non-functional requirements, constraints
- Domain model should capture commonality and variants
- Modular domain model
- Support for multiple levels of abstraction
- Scaleable - the domain model should be able to capture a large volume of domain information
- Implementable - the domain model should provide direct guidelines for design of a generic architecture

Domain modeling: Lessons learnt

- A domain model should:
  - provide guidelines for the design of a generic architecture
  - describe a range of functions and variants implemented within a generic architecture for reuse
- You must be innovative to model domains
- Different domains may require different modeling techniques
- Existing notations can be augmented to model variants
- Different views of a model must be cross-referenced for traceability
- How to make a domain model understandable as the volume of information grows, with complex dependencies among variants?

Product Line Approach

Part II: Generic architectures for Product Lines

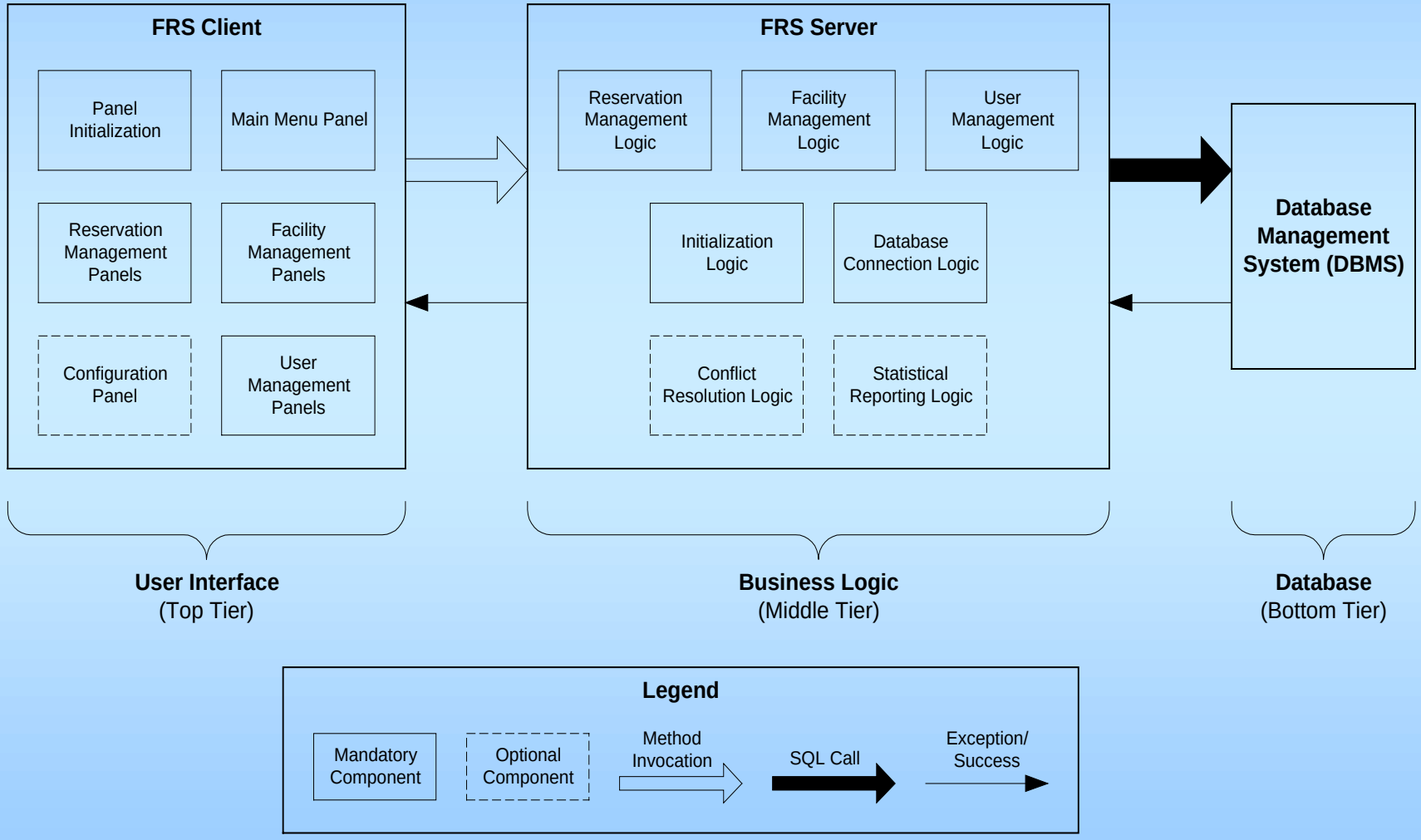
Overview of topics

- Software architectures
  - runtime and construction-time architectures
- Generic architectures for Product Lines
- Techniques for generic architectures:
  - pre-processing, information hiding
  - table-driven techniques
  - OO frameworks
  - application generators
  - Component-based Software Engineering (CBSE)
  - Frame technology
- Generic architecture customization and evolution

Software architecture: an example

- Shaw/Garlan[(Shaw 96)] define architecture as components, connectors and constraints
- this definition addresses primarily areas of concern that matter during runtime:
  - allocation of functions to components, distribution of components, synchronization, control, computational threads
  - component connectors: data communication, invocation of services

An example: FRS runtime architecture



Software architecture - a broader concept

A software architecture plays many roles:

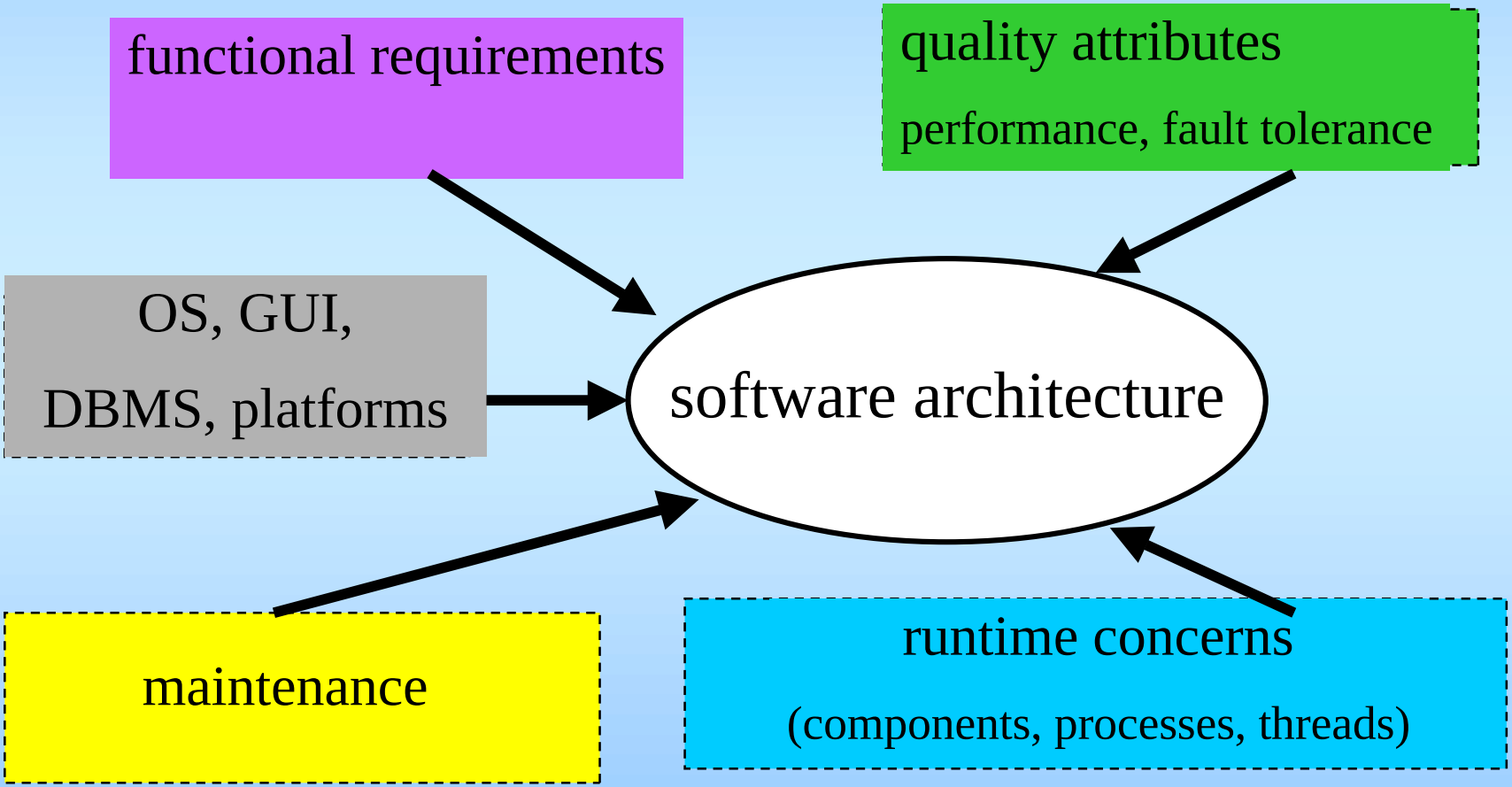
- guides in software development and maintenance
  - what does it take to change a software system?
- tells how a software system satisfies:
  - functional requirements
  - quality attributes such as:
    - performance, reliability, maintainability
    - reusability (in Product Line context)
- describes distribution, synchronization
- facilitates communication among system stakeholders:
  - users, developers and maintainers

Software architecture

- Software architecture: *multiple structures* describing software system in an *abstract way*:
  - in terms of components, visible properties of components and component relationships (Bass 98)
- Each architecture structure addresses a certain concern of some system stakeholders:
  - users, developers and maintainers
- Architecture tries to achieve ***separation of concerns***
  - ability to understand, describe, implement and easily modify a given area of concern, in separation from other areas of concerns.

Software architecture

architecture must satisfy concerns of different stakeholders:

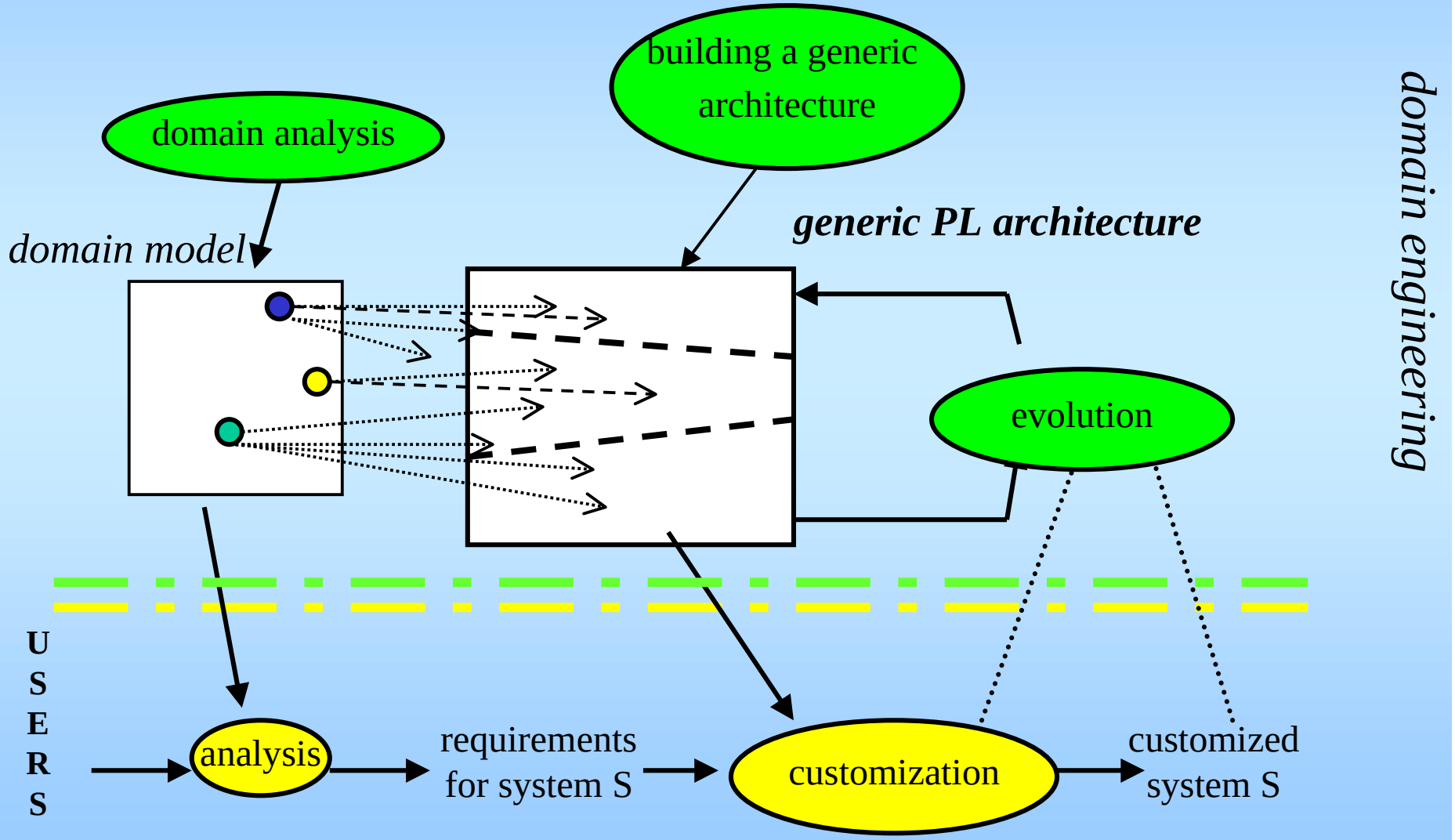


A generic architecture for a Product Line (PL)

A generic PL architecture is like a software architecture but in addition it plays some extra roles:

- defines an architecture shared by PL members
- defines components that can be reused when building PL members
- explains how to customize components to meet required (anticipate) variants
- helps evolve components as we address new unexpected requirements

How to handle variants?



SDLC for Product Lines

- bottom-up or top-down?
- it is difficult to design a generic architecture without having implemented a couple of systems in a domain
- it is difficult to extract a generic architecture based on systems designed in ad hoc way
- Model a typical (default) system (e.g., in UML)
- Design and implement default system in a domain
  - base design on control and data abstractions
- Develop generic architecture incrementally:
  - select some area (subset of system functions) and variants
  - model variants in respect to domain defaults
  - implement a related part of generic architecture
  - separate concerns to decouple components
  - integrate with the rest of the system

A generic PL architecture

Approaches range from:

- informal design and implementation guidelines

to:

- code instrumented with macros
- OO frameworks
- component architectures
- frames (Netron Inc.)
- software packages (such as SAP)
- application generators

Pre-processing, conditional compilation

- improvement over copy and paste
- codes implementing variants are marked with compiler flags and can be selectively included into a program:

```
#define Max 100
# if condition-A
        code for customer A
#else if condition-B
        code for customer B
```

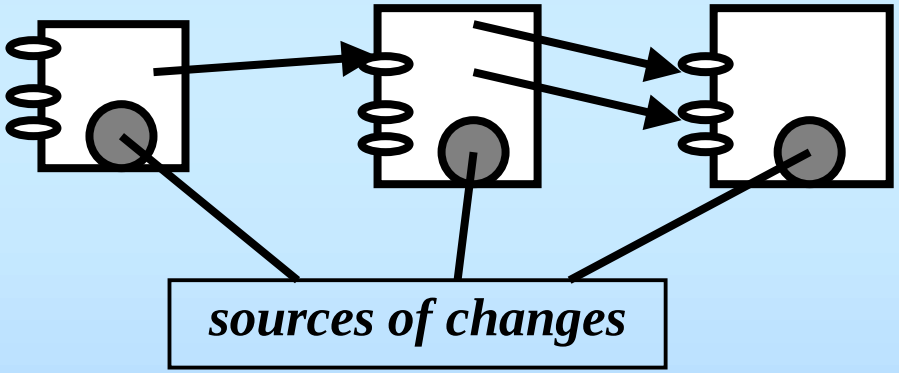
- large granularity variants implemented in separate files: #include “file”
- pre-processor and file configuration management system help manage variants

Source level pre-processing does not scale well:

- failing to utilize design to manage software complexity(karh97)
- changes to common parts are propagated to all points of reuse – but what if we cannot reuse “as is”?
- mappings between variants and relevant codes are complicated
- dependencies among variants are not explicit
- each variant gives raise to a new condition and #if
  - hundreds of interdependent conditions
  - as the number of variants grows, managing variants becomes complex
- source code becomes unreadable, difficult to test

Information hiding, layered design

- identify source of change (variants)
- design modules to encapsulate source of changes:
  - make impact of changes local to a small number of modules
  - hide changes behind stable module interfaces



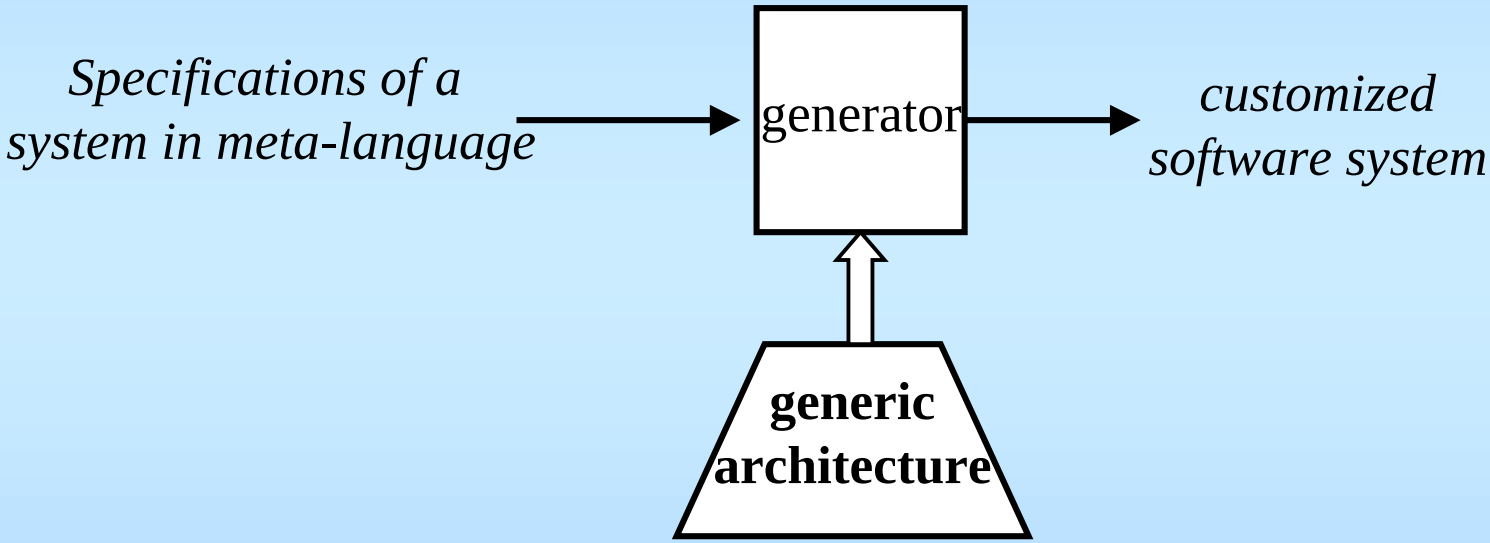
- *identifying right candidate modules and defining module interfaces is critical in this style of design for change*

Generic solutions to common problems

- Analysis and design patterns
- Common, recurring problems at application level:
  - user access right control mechanism
  - problem tracking, task management, failure detection
- Recurring problems at design level:
  - platform hiding module layer
  - OO design patterns
- Techniques for designing generic solutions:
  - data-driven techniques and generation techniques

Generation techniques

- generators address variants during program construction
- variants are described in a domain-specific meta-language



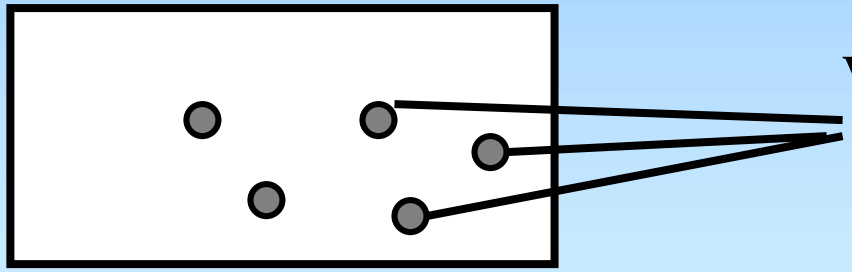
- unlike in a data-driven solution, generator interprets variants and transforms its generic architecture accordingly
- examples: compiler-compilers, 4GL, CASE, GUI, Visual Programming Environments

Generic solutions to common problems: summary

when can we design a generic components?

- we must be able to find a model for variants (analysis or design pattern)
- variants are then defined as an instance of a model:
  - as instances of model entities or relationships (e.g., in RCM)
  - by assigning values to entity attributes or by adding attributes to model entities
  - BNF grammar rules or any other way, depending on a model
- usually, variants are stored in tables that drive a generic component
- there must be clean interface between a generic component and the rest of a system
- see other examples in case studies (LN set #4a and #4b)
- a generic solution to a problem may be reused within a Product Line and across Product Lines

Object-Oriented frameworks



- OO framework is built bottom-up, by refining existing OO systems
- identify hot spots and generalize them
- customization: inheriting from “hot spots”
- use of design patterns

Component-based Software Engineering

- white-box source level components:
  - data structures: array, struct
  - procedures or functions: sine (x), sort (x)
  - class, object: data structure + operations (methods)

    class Stack: array + push(), pop(), top(), etc.
  - groups of classes
  - incomplete, generic components (for flexibility!)

    abstract classes, source code with macros
- black-box binary components
  - MS Office: WORD, Excel, PowerPoint
  - compiled libraries, OS
  - GUI-generated user interface, DBMS
  - in CBSE: Java beans, ActiveX controls, CORBA components

Component definitions(ICSE'20)

Runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime (Gartner Group)

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party. (Szyperski, *Component Software*)

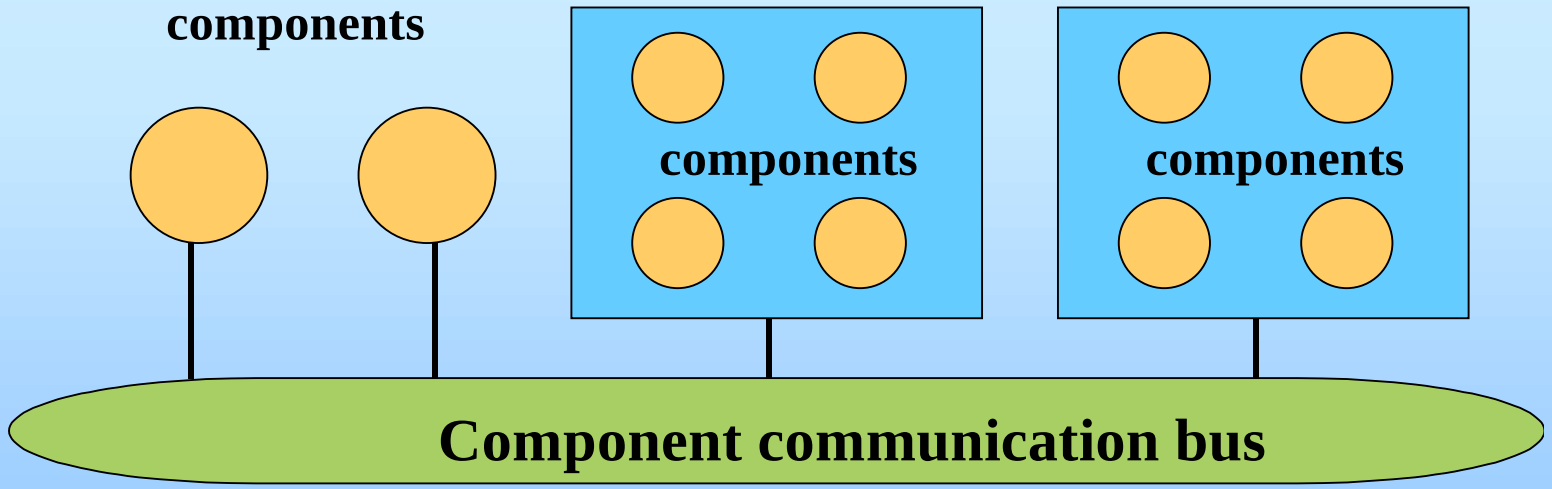
A business component represents the software implementation of an "autonomous" business concept or business process. It consists of all the software artefacts necessary to express, implement and deploy the concept as a reusable element of a larger business system. (SSA)

Component characteristics

- binary unit, independently developed and tested
- explicit interfaces
- unit of composition
  - rapidly assembled in different, scaleable configurations
- replaceable
- Examples of components:
  - the whole applications such as word processor or compiler
  - GUI, DBMS
  - business objects (Customer, Order)
  - beans (EJB), ActiveX controls, CORBA objects

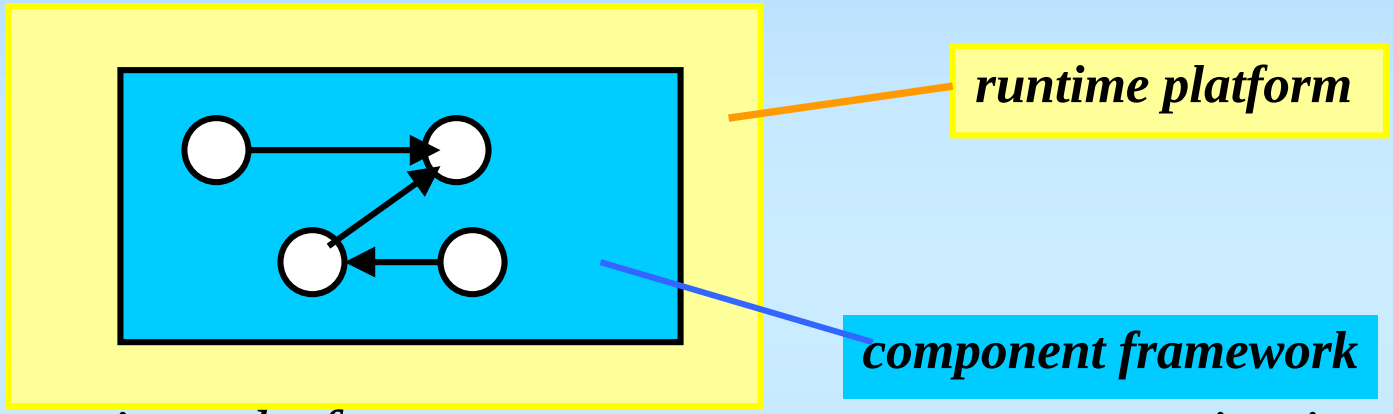
Component-based systems

- a component-based system is built out of collection of executable, cooperating components
- Distributed Component Platforms (DCP) facilitate component interoperability
  - for example, ActiveX, JavaBeans, CORBA implementations



Component-based Software Engineering (CBSE)

- CBSE (1): a vision that software system can be built out of independent binary, vendor provided components

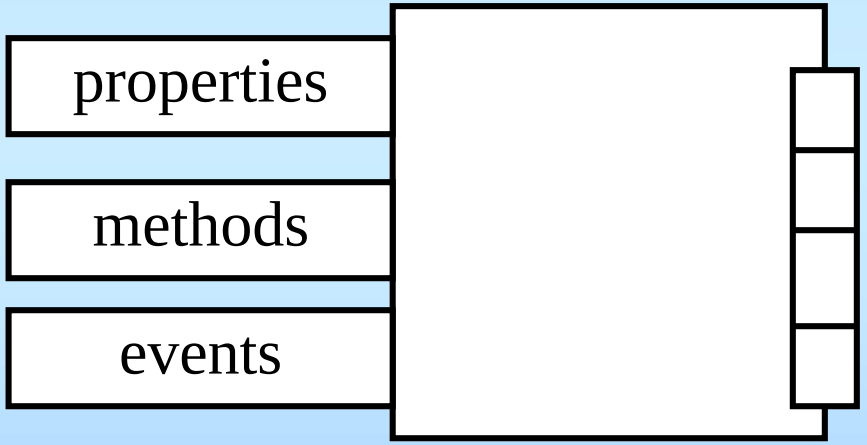


- *runtime platform* supports component communication standards
  - how do components call each other services?
  - how should components' APIs be defined?
- *component framework* provides application-level components (e.g., customer, order, etc.)

Distributed Component Platforms (DCP)

Enterprise JavaBeans, DCOM/ActiveX, CORBA implementations

- dual nature of components:
  - binary, runtime components
  - customizable, construction units



- multiple interfaces
- construction-time view:
  - introspection mechanism to view component interface
  - visual and programatic ways to customize components

DCPs: runtime view

- runtime introspection of component interfaces
- late binding, dynamic linking
  - new components can be added to a system
  - a new version can replace an old component
- message passing via ORB-like mechanism
  - no direct communication path between components
  - components that understand a standard interface language can interact with each other
  - component co-operation in a distributed environment

Indirect communication between components

- remote method invocation
  - a component sending a message does not need know the id nor location of a component that receives a message



- event-based, implicit method invocation

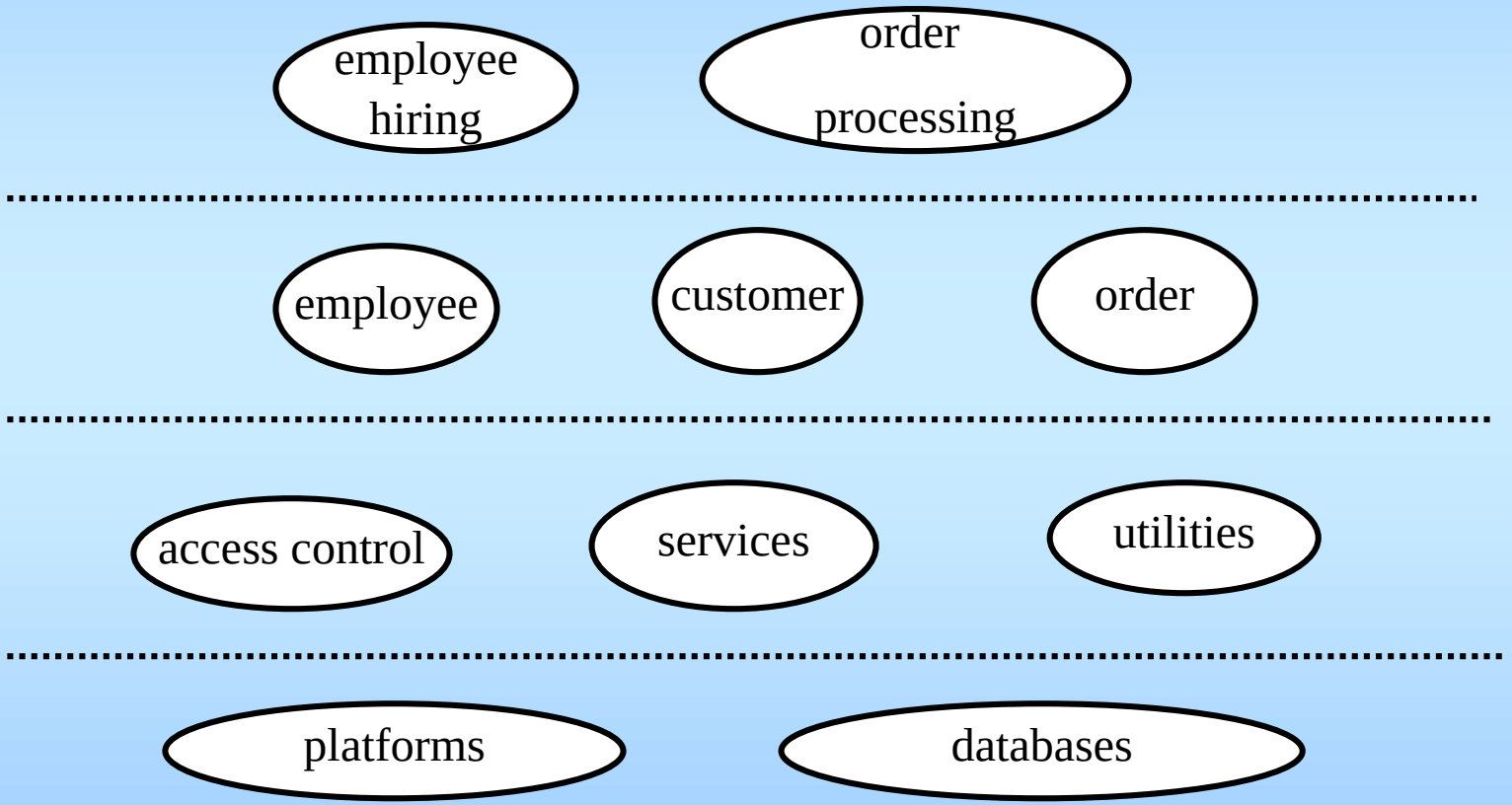


  - the results of component actions are recorded as *events* and handled by event manager
  - other components can register their interest in specific events
  - once an event occurs, an event manager invokes components interested in that event

CBSE: a software design approach

- CBSE (2): a vision that software system can be designed as a set of co-operating independent components
- both binary COTS and components with source code
- expected benefits:
  - no more huge sources and executables, ease of maintenance
  - modularity of requirements
  - reuse, incremental evolution by replacing components
  - easier adaptation to technology changes
- pitfalls and trade-offs of CBSE are not clear yet

Component layers



CBSE: open problems

- methods for cbse are immature
- can i maintain a component in separation from the rest?
- what happens if a change affects many components?
  - how do i know which components are affected and how?
- what will happen during long term evolution?
  - what will happen to components affected by many independent changes?
- which design objectives can be achieved with cbse? what are properties of systems developed with components?
- which types of systems (parts?) will benefit from components?

A generic architecture for FRS Product Line

- given business goals for generic PL architectures:
  - it must be easy to customize a generic architecture during system engineering

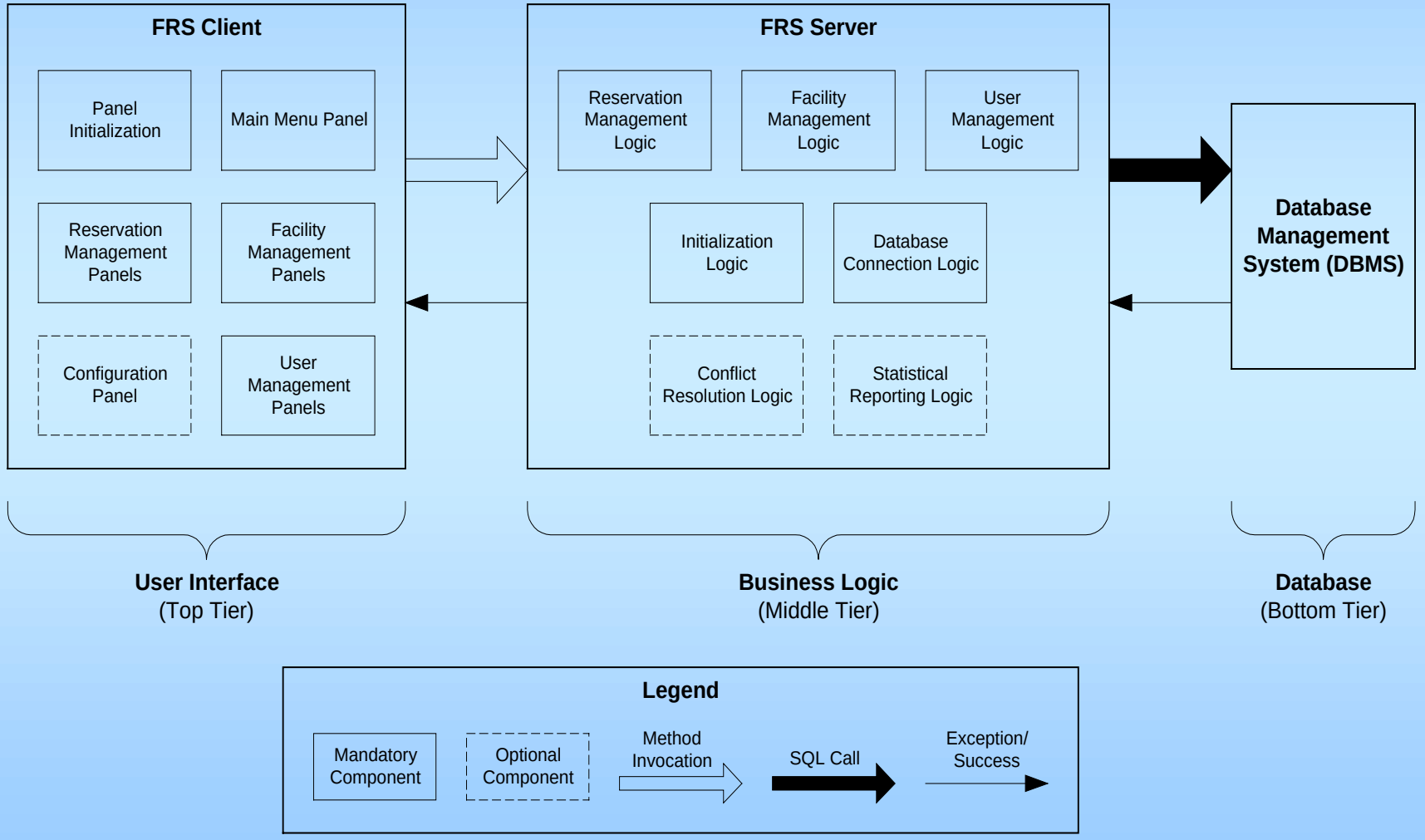
  programmer's productivity will depend on that

  - it must be easy to evolve a generic architecture

  adding new unexpected requirements over years of maintenance, should not overly complicate the architecture

- what should a generic FRS architecture look like?

Component-based generic FRS architecture



Example: customization and evolution of FRS components

I. Consider two anticipated variant requirements (or-features):

- view reservations by facility and/or data
  - anticipated features have been discovered during domain modeling and implemented into a generic FRS architecture
  - during component customization, we should be able to select any of anticipated variant requirements and include into the product

II. Consider new unexpected requirement:

- view reservations by user
  - new unexpected requirements have not been discovered during domain analysis
  - we may want to include new requirement only into a certain product, and/or include it into a generic FRS architecture for future reuse in many FRS products (evolution)

Addressing anticipated variant requirements

1. we can implement all anticipated variant requirements into the relevant components:

```
flags:
        view-by-facility        : BOOL
        view-by-date            : BOOL
methods:
        ReservLst ViewByFacility ( Facility )
        ReservLst ViewByDate ( Date )
```

   – then, both components and systems will grow in size

2. we can produce a separate component version for different each combination of variant requirements

   – then, the number of component versions will grow in size

     pessimistic estimation: if four independent requirements are implemented into a component X, we may have $2^4 = 16$ component versions

Evolving FRS components

suppose we need add a new unexpected requirement:

FRS will allow one to view reservations by the user

components affected by the new requirement:

- "Reservation Management Panels" component:
  - add new option to the menu for viewing reservations
- "Facility Management Logic" component:
  - add logic for the new retrieval method
  - add event handling code for the new retrieval method
  - add a new function that retrieves reservations from the database into the "Facility Management Logic" component

Evolving FRS components

- Modifications affect interfaces of some components
  - a new declaration for a data structure to retrieve reservations for a particular user
  - interface declarations for operations that will allow the FRS client to obtain a list of registered users and reservation data

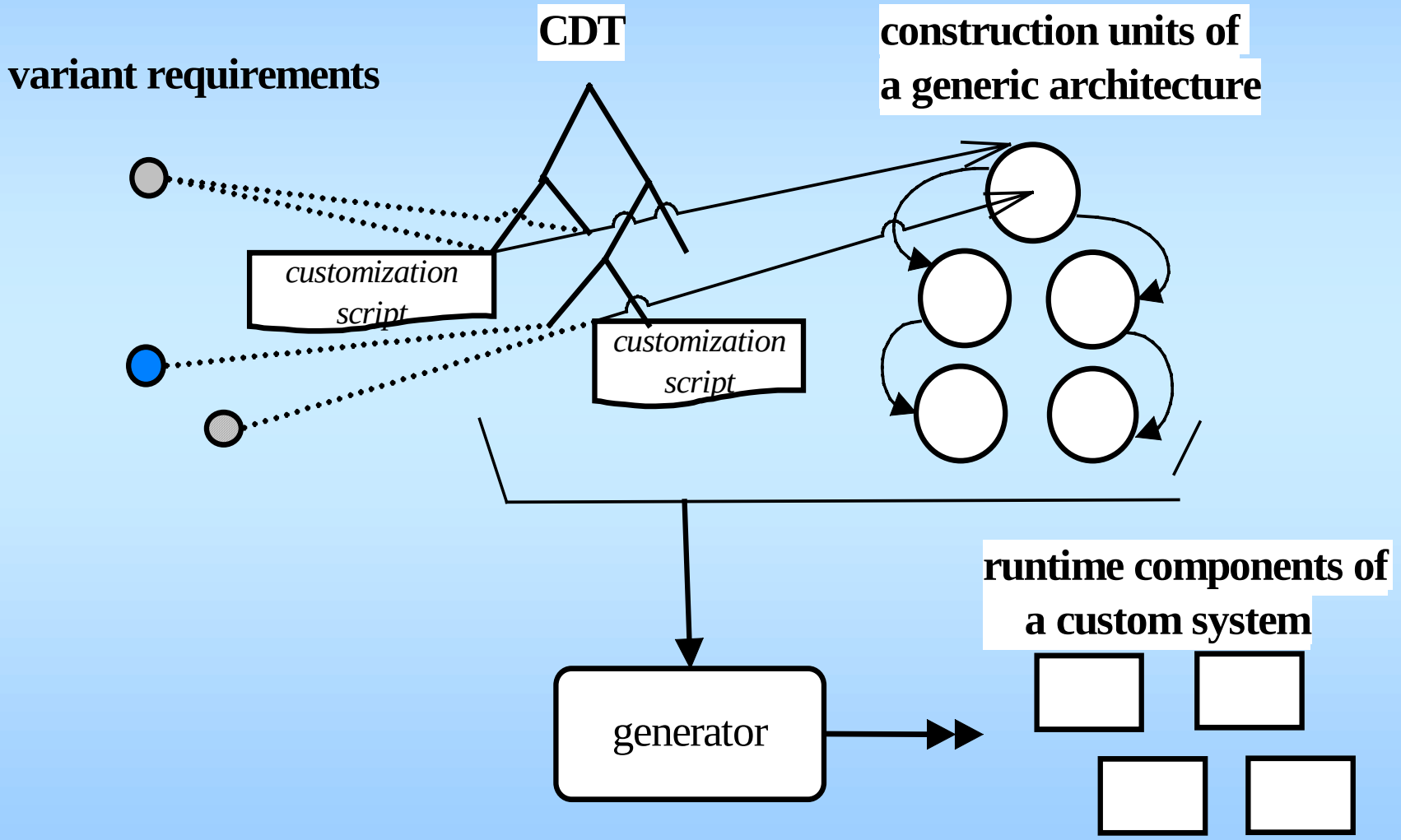
Components cannot be extended in arbitrary ways without source code

- Possible solutions if source code is available:
  - derive new components using inheritance
  - use design patterns to evolve components

Summary of problems

- growing size of components
- growing number of component versions
- growing complexity of the generic architecture
- lack of guidance in understanding the impact of requirements on components of the architecture
- rationale for why components have been modified in certain ways is lost

Component construction environment



Comments

A construction environment is meant for customization and evolution of component-based systems. The construction environment is to alleviate problems identified before.

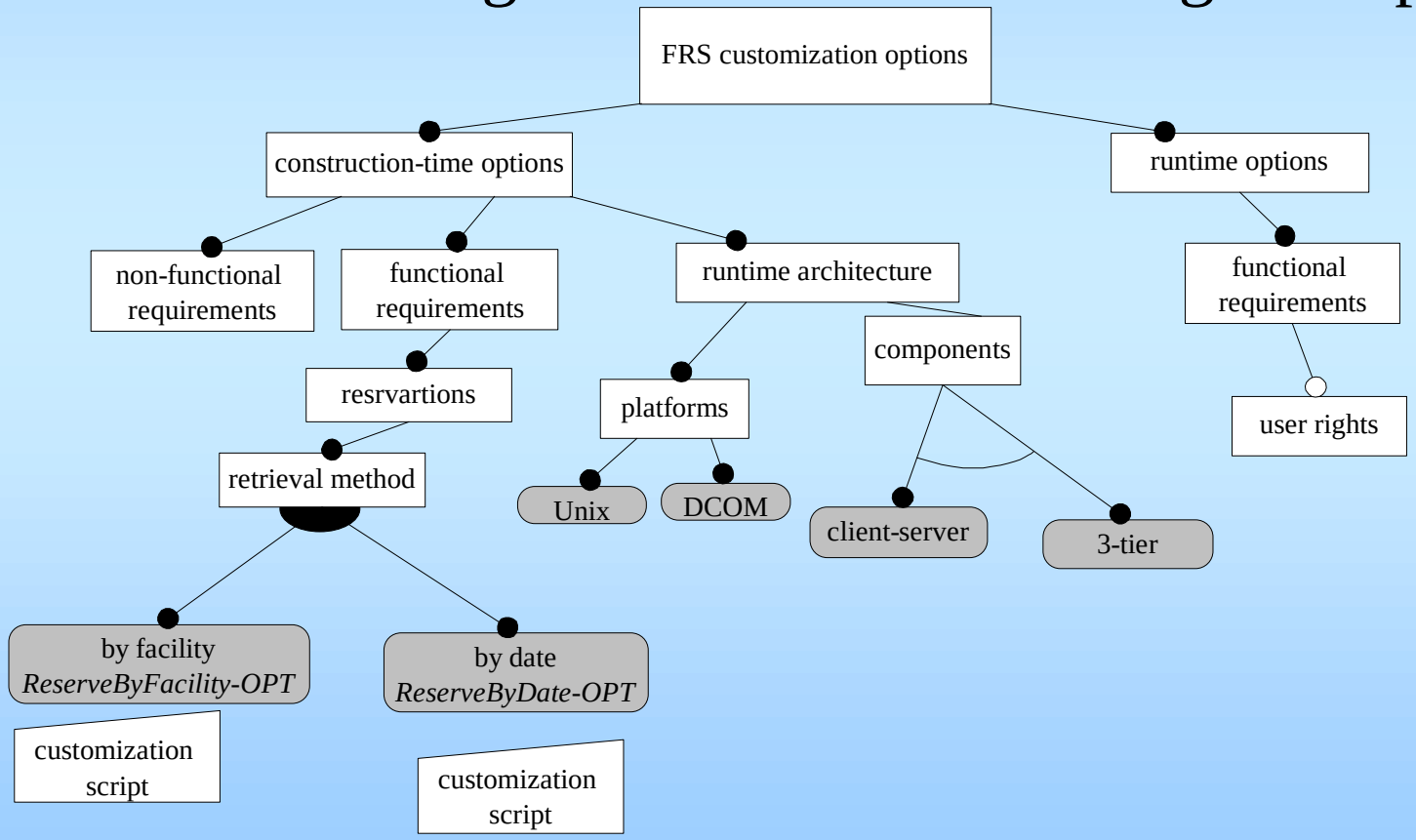
A construction environment includes a generic architecture and two global structures, namely, a domain model and a Customization Decision Tree (CDT). A generic architecture is a hierarchy of construction units from which custom components of a runtime system are produced. The double arrow in the slide represents processing that is required to produce runtime system components from customized construction units of a generic architecture. This includes selection and customization of construction units, generation of custom components, assembling components into a custom system and testing.

The role of a domain model and CDT is to facilitate understanding of variants in a domain, understanding of how variants are reflected in a generic architecture and what customizations are required to incorporate variants into components.

A Customization Decision Tree (CDT) helps understand customizations that lead to satisfying variant requirements. A “customization option” refers to a particular way to customize the generic architecture. A CDT is a hierarchical organization of customization options. At each level of a CDT, there may be either conceptual groupings of customization options or customization options themselves. The hierarchical organization facilitates the visual location of a particular customization option.

Customization Decision Tree (CDT)

- CDT extends feature models with information how to customize a generic architecture for given options



Handling variant requirements

- retrieve reservations by facility type and reservation date

For each variant requirement R you need in the product:

1. “Drill-down” the CDT looking for a variant requirement R
2. Is requirement R consistent with other variant requirements that you need in the product?
3. If yes, select the relevant customization script

Run generator to customize construction units accordingly

Adding new requirements

- New option: “retrieve reservations by the user”

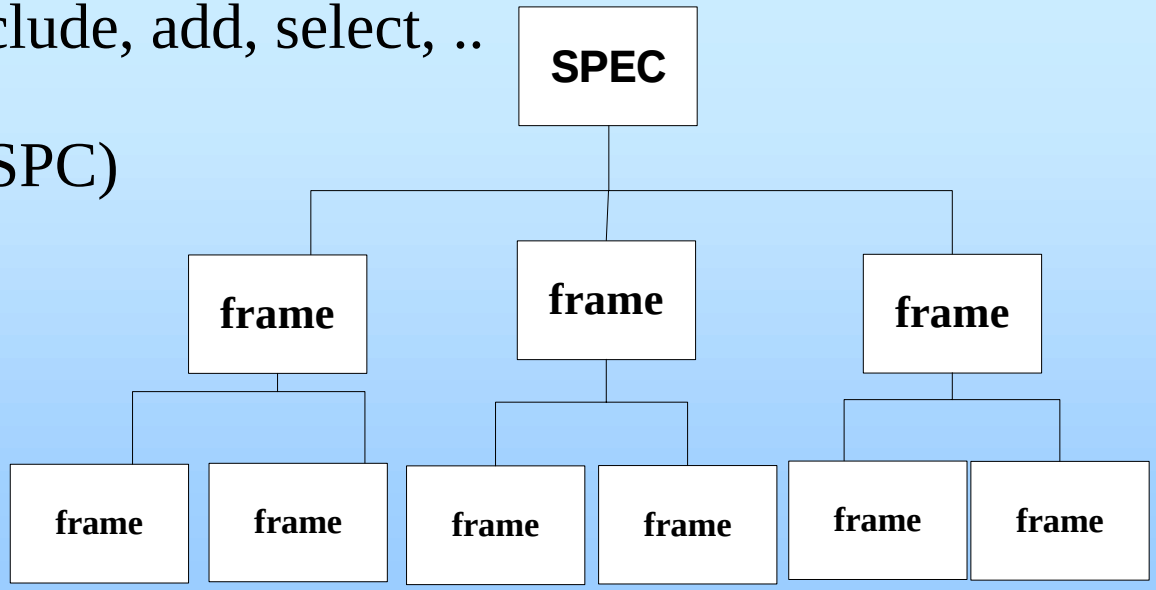
1. Drill-down the CDT looking for the option
2. No such option found under the “Retrieval method” group
3. Study customization scripts for other options in this group
4. Are there any conflicts between the new option and other requirements?

5a. Write a customization script for the new option

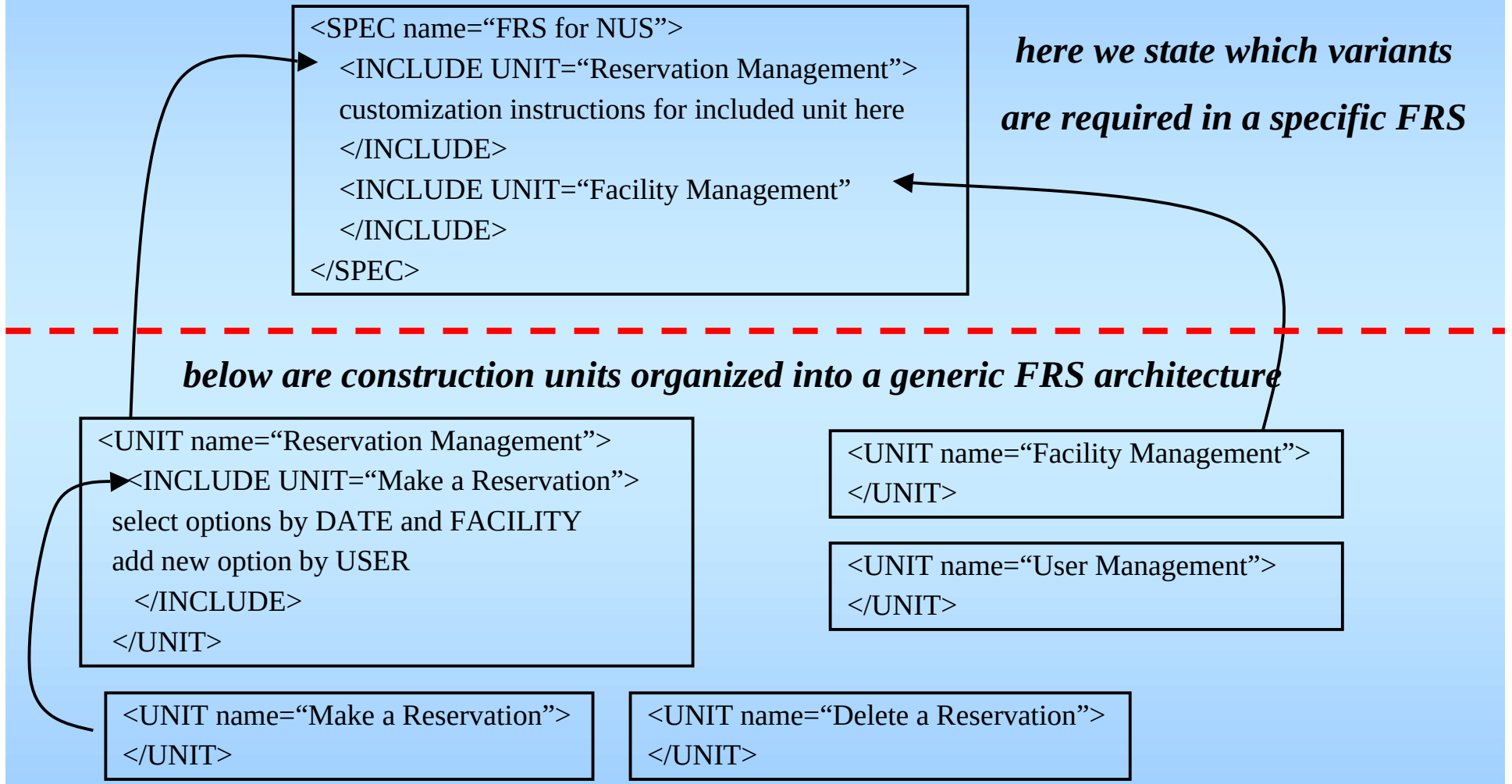
5b. Incorporate the new option into the generic architecture

A generic architecture: an example

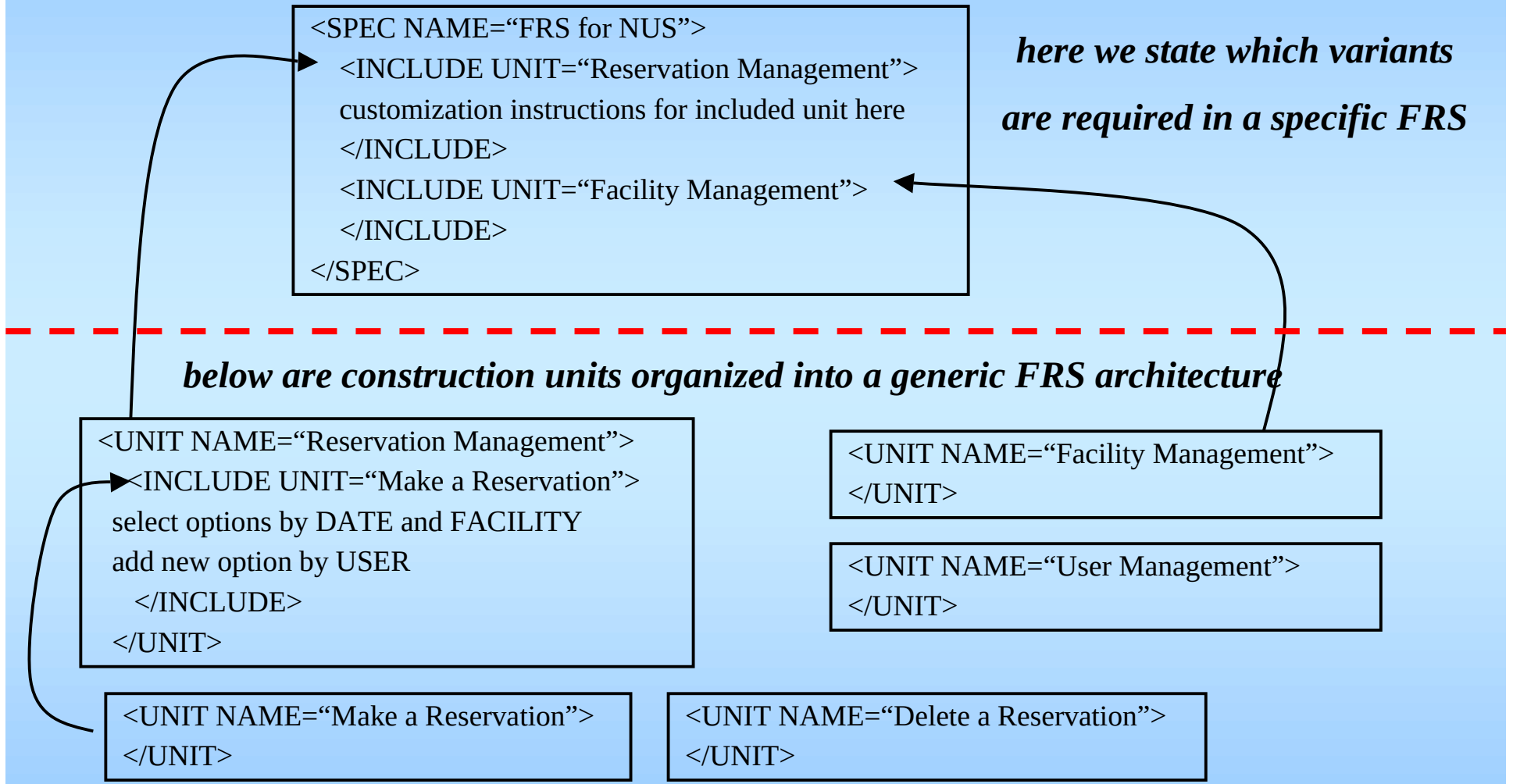
- frame technology(Bassett 97) by Netron, Inc·
- generic architecture: a hierarchy of frames
- frame: generic construction unit with break points
- frame commands: include, add, select, ..
- specification frame (SPC)
- frame processor



Frames as hierarchy of XML code documents



Frames as hierarchy of XML code documents



Customization examples

```
<SPEC NAME="FRS for NUS" >
   <SET VAR="FRS_Type" VALUE="Academic" />
   <SET VAR="Version" VALUE="1.02" />
   <SET VAR="Payment_option" VALUE="FALSE" />
   <INCLUDE UNIT="Reservation Management">
        <SET VAR="ViewRes_options" VALUE= "FACILITY ,DATE " />
        <INSERT-AFTER BREAK="VIEW_RES_METHOD" >
              …  view reservations by USER …
         </INSERT-AFTER>
   </INCLUDE>
   <INCLUDE UNIT="Facility Management">
     …..
   </INCLUDE>
</SPEC>
```

```
<UNIT NAME="Reservation Management">
   <INCLUDE UNIT="Make a Reservation">
     select options by DATE and FACILITY
     add new option by USER
   </INCLUDE>
</UNIT>
```

```
<UNIT NAME="Make a Reservation">
   <INCLUDE UNIT="View Reservations">
   </INCLUDE>
</UNIT>
```

Customization commands

XML command tags	The meaning
<INCLUDE UNIT="*name_of_unit* > customization commands </INCLUDE>	The specified UNIT is included after customization when the <INCLUDE UNIT> tag is encountered.
<INSERT-BEFORE BREAK="*breakpoint*"> .. code to be inserted </INSERT-BEFORE> <INSERT-AFTER BREAK="*breakpoint*"> .. code to be inserted <INSERT-AFTER BREAK="*breakpoint*"> <INSERT BREAK="*breakpoint*"> </INSERT>	The specified code is inserted before/after/at breakpoint.
<BREAK NAME="*breakpoint*"> </BREAK>	Marks a breakpoint in a construction UNIT.
<SET VAR=*variable_name* VALUE="*value*" />	Assigns value to a variable. Variables are used to control customizations.
<VAR NAME="*variable_name*">	Marks a location where the variable value assigned by <SET> is inserted.
<SELECT OPTION= "*variable*"> <OPTION VALUE= "*value*"> ..</OPTION> <OPTION VALUE="*value*"> ..</OPTION> <OTHERWISE/> </SELECT>	Selects code based on the *value* of the *variable*

XML construction unit: View reservations

```
<UNIT NAME=”View Reservations” >
class ViewRes extends FRSPanels
{               viewResMethodList = new List ();
                viewResMethodList.addItem (“Reservation ID”); // default
// anticipated variants defined here
<SELECT OPTION=“ViewRes_options” >
  <OPTION=“FACILITY”>  // by Facility
    viewResMethodList.addItem (“Facility”);
  </OPTION>
  <OPTION=“DATE” >  // by Date
    viewResMethodList.addItem (“Date”);
  </OPTION>
</SELECT>
<BREAK NAME=“VIEW_RES_METHOD” > // any further unexpected variants
…
</BREAK>
// event handling methods:
<BREAK NAME=“VIEW_RES_METHOD_EVENT” >
… other codes here
</BREAK>
} // End of ViewRes class
</UNIT>
```

Customization SPEC and the result

```
<SPEC NAME=”Select variants to view reservations by Facility and Date”>
<INCLUDE UNIT=”View Reservations” >
  <SET VAR=“ViewRes_options”  VALUE= “DATE, FACILITY” />
</INCLUDE>
 …
</SPEC>
```

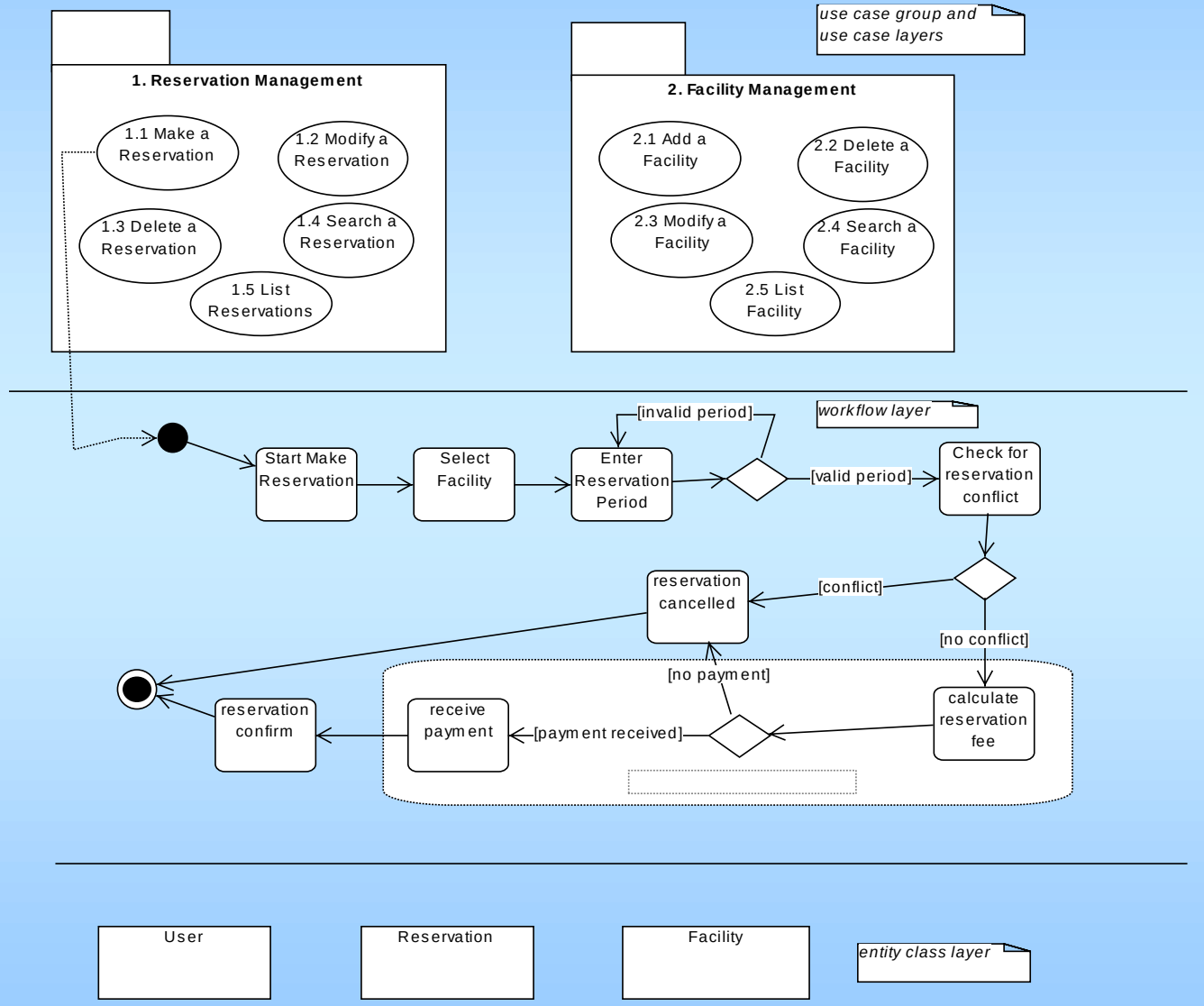
The result of customization:

```
class ViewRes extends FRSPanels
{           viewResMethodList = new List ();
viewResMethodList.addItem (“Reservation ID”); // default
// selected variants:
viewResMethodList.addItem (“Facility”);
viewResMethodList.addItem (“Date”);
} // End of ViewRes class
```

New requirement: view reservations by User

```
<SPEC name="View reservations by User" >
        <INSERT-AFTER BREAK="VIEW_RES_METHOD" >
                viewResMethodList.addItem ("User");
        </INSERT-AFTER>
        <INSERT-AFTER BREAK="VIEW_RES_METHOD_EVENT" >
            } else if  (viewResMethodList.getSelectedItem () == "User"){
                .....
              };
        </INSERT-AFTER>
</SPEC>
```

Separation of functional concerns: example



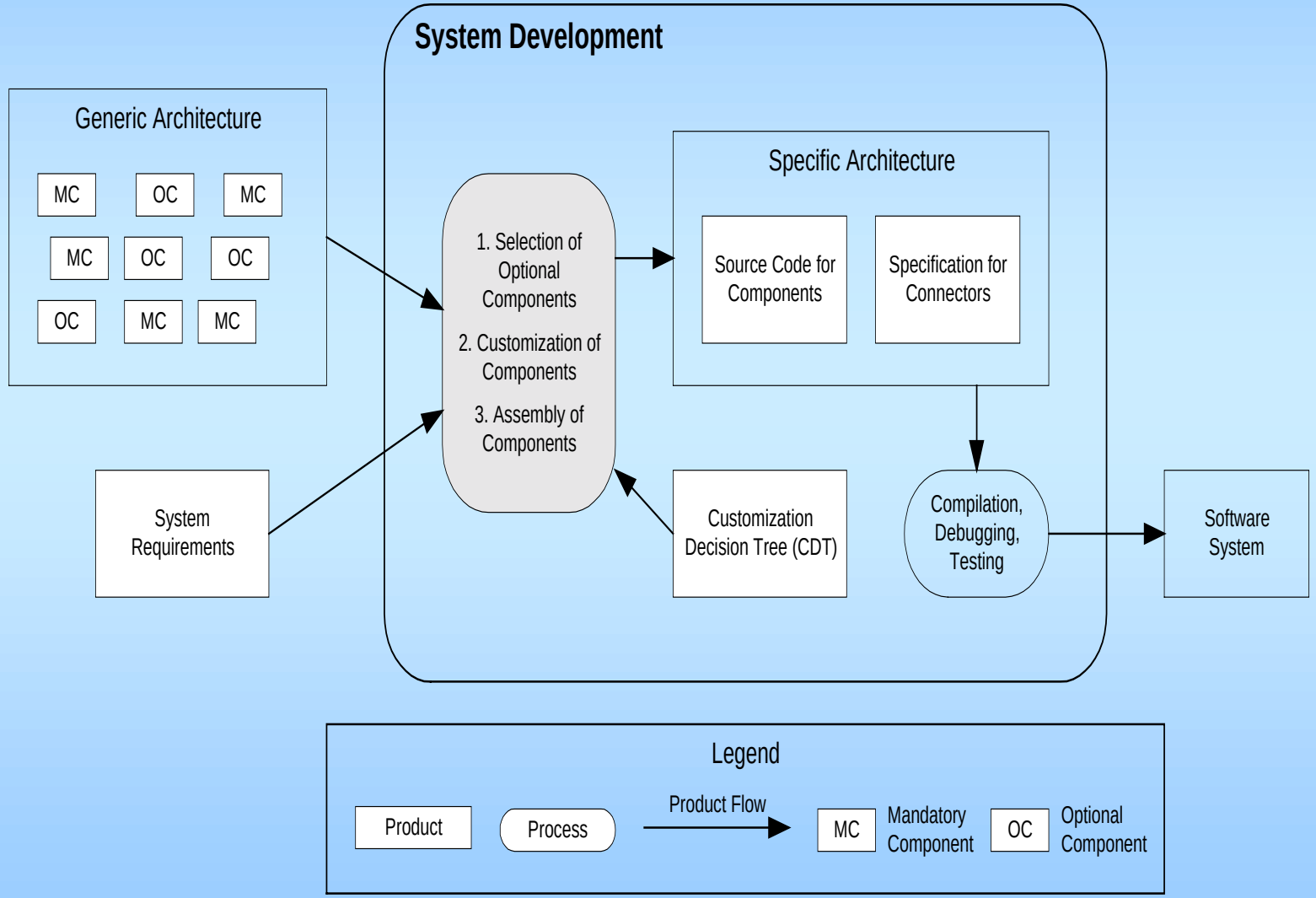
Highlights of the solution

- Mapping variant requirements to relevant architecture construction units
  - a domain model
  - a Customization Decision Tree (CDT)
- Levels of customization
  - at the architecture level first and at the code level next
- Separation of concerns
  - codes related to different areas of concern should be defined in separate layers of construction units
- Composition operation
  - to combine construction units in order to produce runtime components that have required properties

Highlights of the solution, cont.

- Focusing on customization process that leads to satisfying variant requirements rather than on customized components
  - customized components can be produced on demand, we do not have to store and manage them
  - knowledge of how to deal with anticipated variant requirements will help developers understand how to evolve a generic architecture
- Human-oriented architecture
  - mappings between variant requirements, generic architecture and customization process
- Customization tools: a generator

Levels of architecture customization



Challenges in design of generic architectures

- how to achieve separation of concerns?
  - manage each area of concern separately from others
  - combine variants from different areas of concern into a product
  - detect and remove (or resolve) conflicts during customization
- automation of routine customizations
- how can we assure product correctness (and other quality attributes) after customizations?

Conclusions

- There is a wide range of techniques for designing generic architectures - select one that fits the problem in hand
- The choice of the technique will determine ease of architecture customization (i.e., ease of system development) and ease of architecture evolution.
- Generators can simplify system development by automating routine customizations
- A generic PL architecture should be open to developers to facilitate architecture evolution

References

Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998

Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997

Bosch, J. *Design & Use of Software Architectures – Adopting and evolving a product-line approach*, Addison-Wesley, 2000

Brown, A. and Wallnau, K. "The Current State of CBSE," *IEEE Software*, September/October, 1998, pp. 37-46

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996]

Digre, T. "Business Object Component Architecture," *IEEE Software*, September/October 1998, pp. 60-69

Fowler, M. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*, Addison-Wesley, 1994

Jacobson, I., Griess, M. and Jonsson, P. *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley Longman, 1997, pp.39-42.

Johnson, R. and Foote, B. "Designing Reusable Classes," *Journal of Object-Oriented Programming*, June 1988, Vol.1, No.2, pp. 22-35.

References

Karinhen, A., Ran, A. and Tallgren, T. "Configuring designs for reuse,"*Proc. ICSE'97*, Boston, May 1997, pp. 701-710

Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. "Aspect-Oriented Programming," *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, Springer-Verlag LNCS 1241. June 1997.

Krieger, D. and Adler, R. "The Emergence of Distributed Component Platforms," *IEEE Computer*, March 1998, pp. 43-5]

Macala R., Stuckey, L. Jr. and Gross, D. "Managing Domain- Specific, Product-Line Development," *IEEE Software*, May 1996, pp. 57- 67Lewis Ted et al *Object-Oriented Application Frameworks*

Orfali, R and Harkey, D. *Client/Server Programming with Java and CORBA*, John Wiley and Sons, 1998, second edition

Parnas, D. "On the Design and Development of Program Families," *IEEE Trans. on Software Eng*., March 1976, p. 1-9

Parsons, J. and Wand, Y. "Using Objects for System Analysis," CACM, vol. 40, no. 12, Dec. 1997, pp. 104-110

Shaw, M. and Garlan, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hal, 1996

References

Sommerville, I. and Dean, G. "PCL: a language for modelling evolving system architectures," *Software Engineering Journal*, March 1996, pp.111-121

Szyperski, C. Component Software – Beyond Object-oriented Programming, ACM Press, Addison-Wesley, 1998

Wegscheider, E. “Toward Code-Free Business Application Development,” IEEE Computer, March 1997, pp. 35-43

Voas, J. “Challenges of Using COTS Software in Component-based Development,” *IEEE Computer*, June 1998, pp. 44-52