EAST WEST UNIVERSITY

CSE 412

Software Engineering

Yasin Sazid
Lecturer
Department of CSE
East West University

Topic 5: Software Reuse

Reuse-Based Software Engineering

- A strategy where software development emphasizes reusing existing software
- Motivation:
  - Lower costs
  - Faster delivery
  - Improved quality
  - Better ROI on software assets

Levels of Software Reuse

- **Object and Function Reuse**: Reuse of single functions or classes from libraries
- **Component Reuse**: Reuse of subsystems to objects
- **Application System Reuse**: Entire systems reused or configured for new customers
- **Concept Reuse**: Reuse of design ideas or patterns (e.g., models, patterns)

Benefits of Software Reuse

- **Increased dependability**: Tried-and-tested components
- **Reduced process risk**: Known component costs
- **Effective use of specialists**: Reusable expertise
- **Standards compliance**: Uniform UI and reduced user error
- **Accelerated development**: Faster delivery and validation

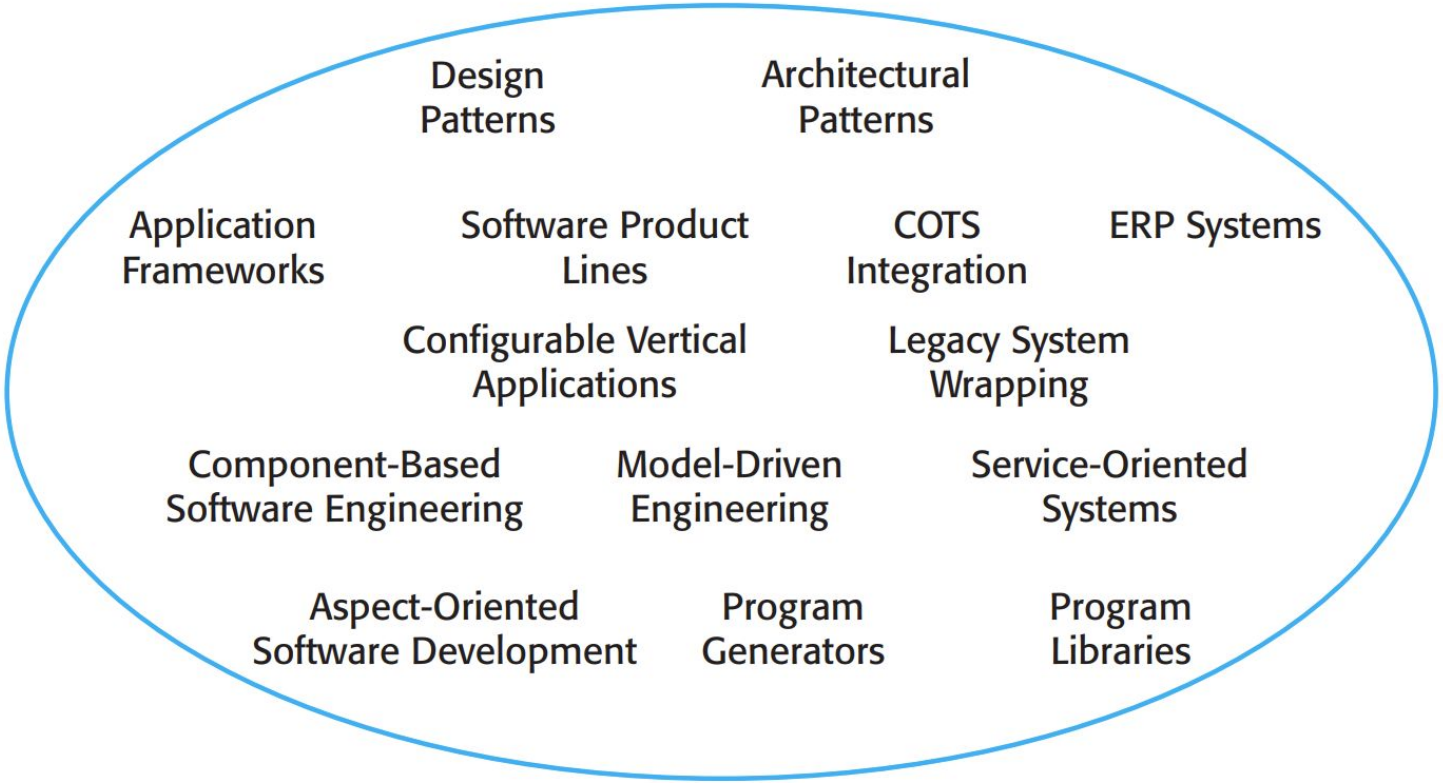
Challenges of Software Reuse

- **Higher maintenance costs**: If source code is unavailable.
- **Tool limitations**: Some tools don’t support reuse.
- **Not-invented-here syndrome**: Reluctance to trust external components.
- **Component library management**: Costly to create and maintain.
- **Component discovery & adaptation**: Time-consuming and complex.

Reuse-Centric Development Process

- Process adaptation needed:
  - Refine requirements to fit reusable assets
  - Design & implementation include component search/evaluation
- Organization-wide reuse programs:
  - Build and maintain reusable assets
  - Adapt processes to enforce reuse

The Reuse Landscape



Key Reuse Considerations

- **Development schedule**: Favor large-grain reuse for fast delivery
- **Expected lifetime**: Consider maintainability and source code availability
- **Team expertise**: Align with familiar reuse tech
- **System criticality**: Certification and performance may limit reuse
- **Application domain**: Prefer domain-specific reusable solutions
- **Target platform**: Ensure compatibility with component models (e.g., .NET)

From Object Reuse to Frameworks

- Early object-oriented development promised object reuse
- But objects are often too small and application-specific
- Adapting an object can be harder than rewriting it
- Solution: Reuse at a larger granularity — via frameworks

Application Frameworks

What is a Framework?

- An integrated set of software artefacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications
- Frameworks:
  - Offer generic structure for applications
  - Let developers extend them for specific needs
  - Enable design reuse, not just class reuse
- Examples: Django, Angular, Flutter, .NET, Spring Boot etc.

Framework Characteristics

- Provide a skeleton architecture and predefined object interactions
- Written in object-oriented languages like Java, C++, C#, Python, Ruby
- Can include other frameworks to support various application parts
- Extended by subclassing or defining callback (hook) methods

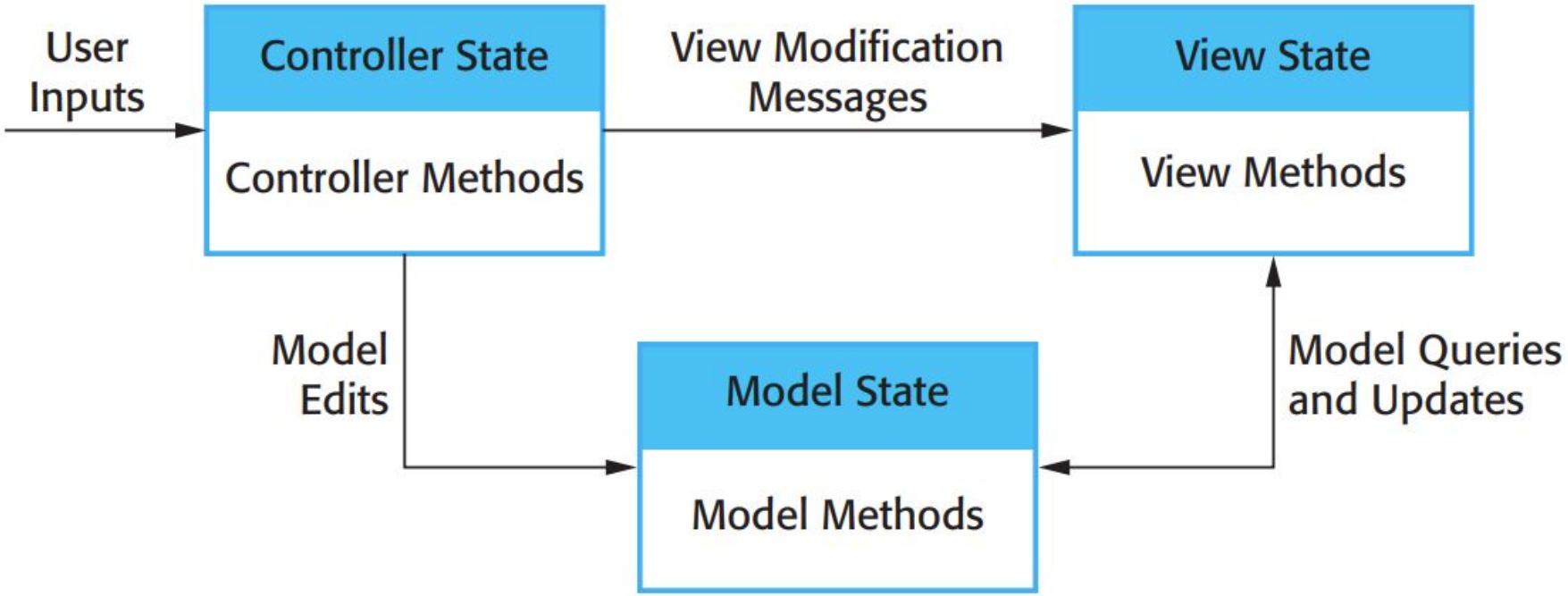
Types of Frameworks

- System Infrastructure Frameworks
  - Support system-level software like GUI, compilers, networking
- Middleware Integration Frameworks
  - Enable component communication and data exchange
  - Examples: .NET, Enterprise Java Beans (EJB)
- Enterprise Application Frameworks
  - Target domain-specific needs (e.g., banking, telecom)
  - Contain embedded domain knowledge
- Web Application Frameworks (WAFs)

Web Application Frameworks (WAFs)

- Popular for building dynamic websites
- Based on MVC pattern (Model-View-Controller):
  - Model: Application state/data
  - View: User interface
  - Controller: Handles user input, updates model/view
- Enables separation of concerns
- Facilitates multiple views and interactions

Model-View-Controller



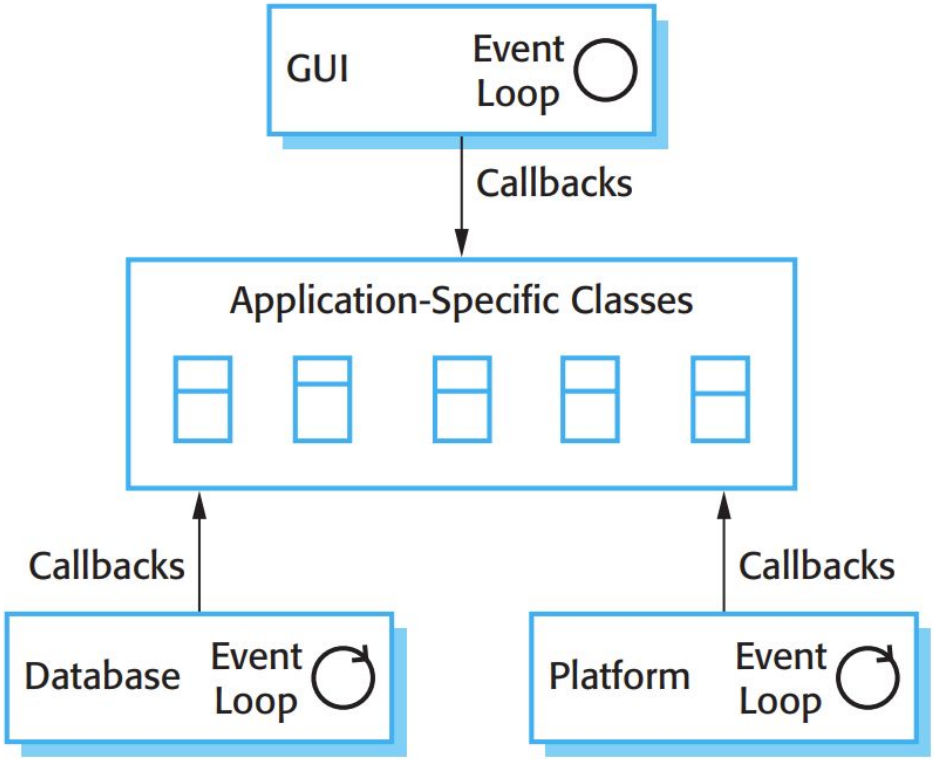
Key Features of WAFs

- Security: Classes for authentication & access control
- Dynamic Pages: Templates + data-binding
- Database Support: Abstract access to databases (e.g., MySQL)
- Session Management: Track user sessions
- AJAX Support: For dynamic, interactive UIs

Inversion of Control (IoC) in Frameworks

- Control flow is managed by framework, not the application
- Developer provides callback (hook) methods
- Framework responds to events (e.g., mouse click) → triggers user-defined code
- This is how **frameworks are different to libraries**
  - A library is a collection of functions, classes, or tools you can use on demand
  - You are in control: you call the library when you need it
  - Libraries are plug-and-play – you pick what you need

Inversion of Control (IoC) in Frameworks

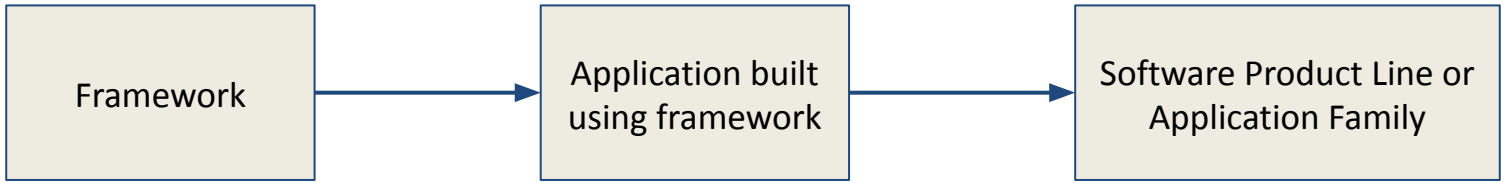


Challenges of Using Frameworks

- Learning curve: May take months to master
- Evaluation: Time-consuming to select the right framework
- Debugging difficulties: Understanding framework internals is hard
- Tool limitations: Debugging tools may not explain framework behavior well

Frameworks & Software Product Lines

- Apps built with frameworks can evolve into software product lines
- For example, using a web framework to build a help desk platform, then customizing it for different clients

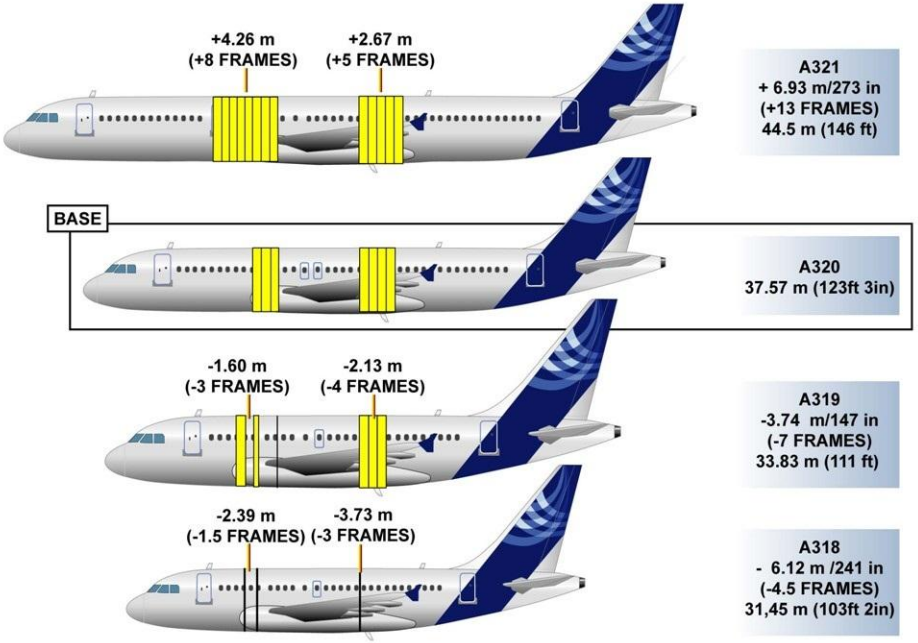


Software Product Line

Few Systems Are Unique

Most organizations produce families of similar systems, differentiated by features.

Few Systems Are Unique



Most organizations produce families of similar systems, differentiated by features.

Software Product Lines

- A software product line (SPL) is a set of related software systems that
  - Share a common architecture and components
  - But are specialized to meet the requirements of different customers or market segments
- Built from a core system that is designed to be configured, adapted, and extended

Benefits of SPL Approach

- High code reuse across applications
- Shortened learning curve for new developers
- Simplified testing via reusable test cases
- Faster development cycles due to reuse of tested components

Evolution of Software Product Lines

- Often begins with informal reuse of code from previous applications
- Over time, managing changes across versions becomes difficult
- Eventually, the organization decides to build a generic, reusable product line architecture

CelsiusTech:Ship System 2000

A family of 55 ship systems

- Integration test of 1-1.5 million SLOC requires 1-2 people.
- Rehosting to a new platform/OS takes 3 months.
- Cost and schedule targets are predictably met.
- Performance/distribution behavior are known in advance.
- Customer satisfaction is high.
- Hardware-to-software cost ratio changed from 35:65 to 80:20.

Nokia Mobile Phones

Product lines with 25-30 new products per year

Across products there are

- varying number of keys
- varying display sizes
- varying sets of features
- 58 languages supported
- 130 countries served
- multiple protocols
- needs for backwards compatibility
- configurable features
- needs for product behavior change after release

Commercial Examples

Successful software product lines have been built for families of

- mobile phones
- command and control ship systems
- ground-based spacecraft systems
- avionics systems
- command and control/situation awareness systems
- pagers
- engine control systems
- billing systems
- web-based retail systems
- printers
- consumer electronic products
- acquisition management enterprise systems

Feature Diagrams

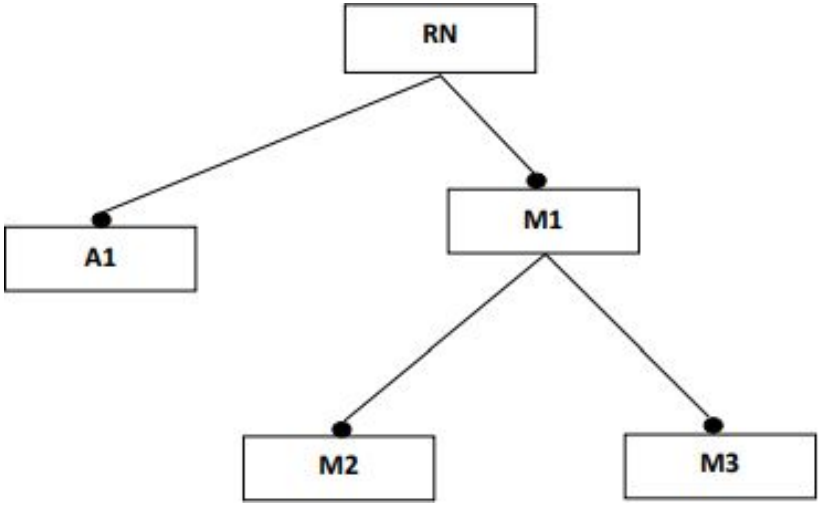
- A feature diagram is a graphical notation that visually represents a feature model, the most important output product of performing domain analysis for product lines.
- The main feature categories provided in the majority of feature diagrams' representations include:
  - Mandatory
  - Alternative
  - Optional
  - AND
  - OR

Mandatory Features

- A mandatory feature is a feature that is always included in the final product of a product line. This means that a mandatory feature is the feature that appears in all possible configurations of a certain product or system.
- A mandatory feature can be a parent or child feature. A child feature is considered mandatory if, and only if, its parent is mandatory and included in the final product.

Mandatory Features

- In the figure, all features are mandatory (parents and children)
- Thus, every instance of the concept node RN will be described by the set {RN, A1, M1, M2, M3}, which is the only possible configuration available considering that the diagram has no variation points

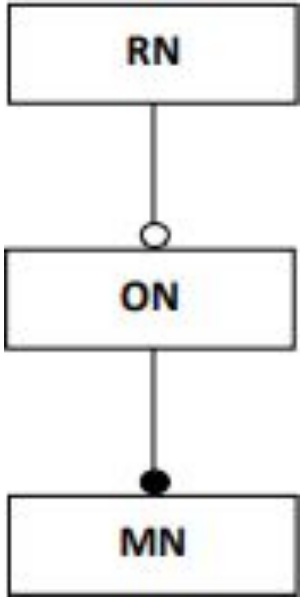


Optional Feature

- An optional feature is a feature that may or may not be selected in the final product of a product line or in the instance description of a concept node.
- The optional feature might be included if its parent is included. If its parent is optional and not included, then the optional feature will not be included.

Optional Feature

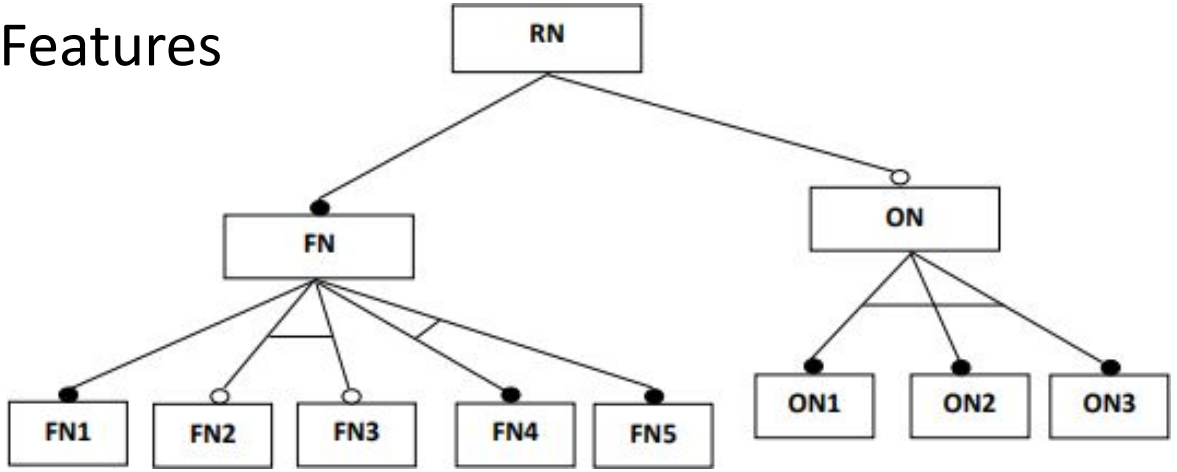
- In this case, the optional feature is a composite feature that has mandatory feature MN as a child feature
- In the instance description of the concept, the RN node is always included, while the mandatory feature MN is dependent on the selection of its parent node ON. If ON is selected, then MN will be selected.
- Therefore, the instance description of the concept will be {RN, ON, MN} or just the concept node {RN}.



Alternative (“one-of”) Features

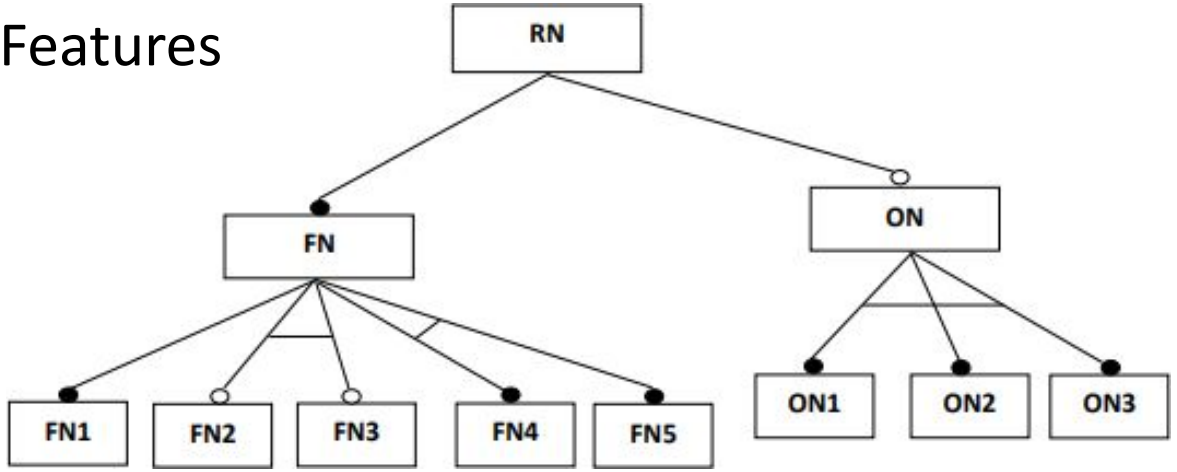
- The alternative feature (also called exclusive “one-of” choice indicates a set of features in which only one feature is selected from the set and included in the final description of the concept (if the parent of the set is also included)

Alternative Features



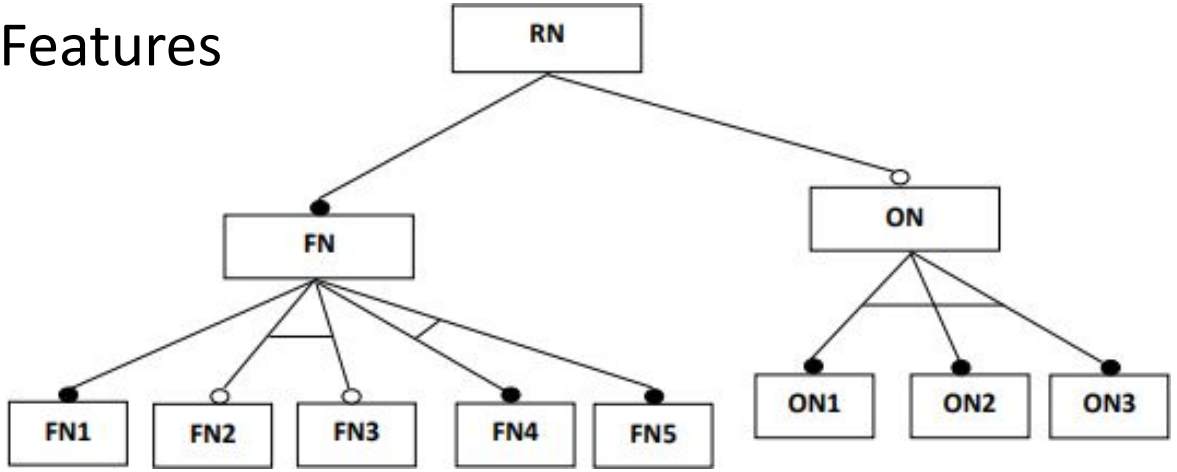
- The feature diagram in the above figure shows concept node RN that has two child features: FN as a mandatory feature and ON as an optional feature.
- The FN feature has one atomic mandatory feature, FN1, and two sets of alternative sub-features {FN2, FN3} and {FN4, FN5}.
- FN1 will be selected since it's a mandatory feature that has a mandatory parent FN, which also has a mandatory parent RN.
- The first alternative sub-features set, FN2 and FN3, contain two optional features. If an alternative set contains only optional features, at most one feature should be chosen.

Alternative Features



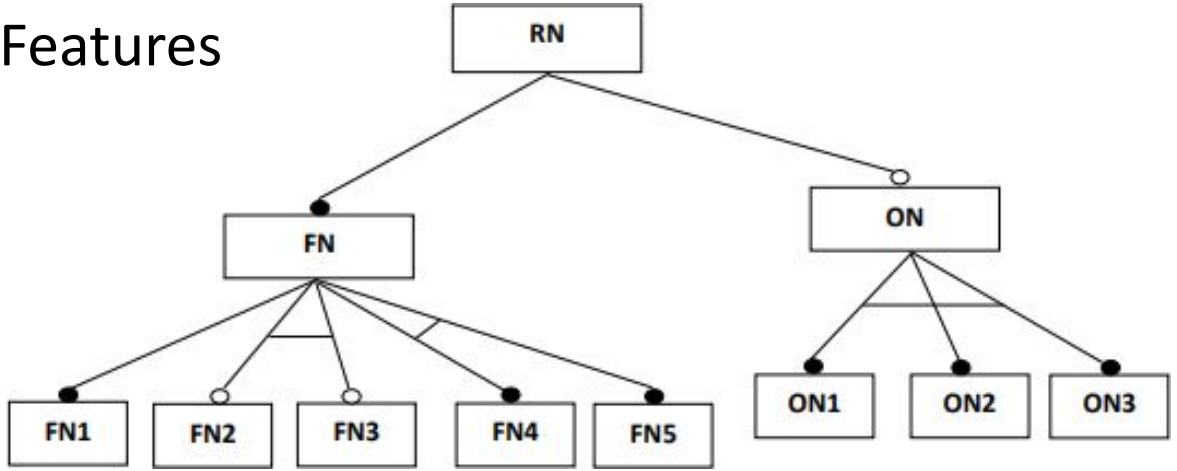
- Other set of alternative sub-features is {FN4, FN5}, in which both are mandatory features
- Since their parent FN is mandatory, exactly one feature should be chosen from the set following the rules of alternative.
- Thus, the number of possible instances from the FN node is {RN, FN, FN1, FN4}, {RN, FN, FN1, FN5}, {RN, FN, FN1, FN4, FN2} , {RN, FN, FN1, FN4, FN3}, {RN, FN, FN1, FN5, FN2}, {RN, FN, FN1, FN5, FN3},

Alternative Features



- The second child feature of the concept node is the optional feature ON that has one set of alternative sub-features containing three mandatory features ON1, ON2, and ON3.
- Although only one feature will be selected from the set, the selected feature might not be included in the instance description because its parent ON is an optional node.
- Thus, the number of possible instances from the RN through ON node is {RN}, {RN, ON, ON1}, {RN, ON, ON2}, {RN, ON, ON3}.

Alternative Features

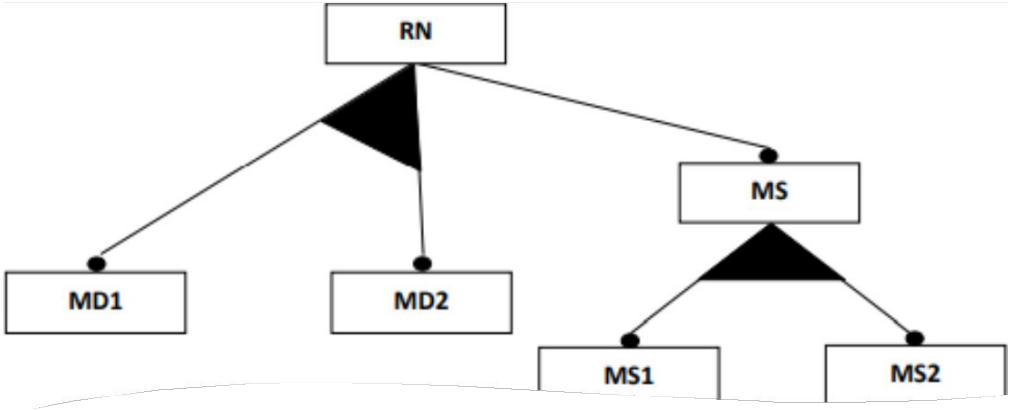


- Multiplying the possible configurations of each node in the feature diagram produces all possible instances for the entire diagram. In this case there are 24 possible instances: 4 (ON) × 1 (FN1) × 3 (FN2, FN3) × 2 (FN4, FN5) = 24.
- The 24 possible instances are:
- {RN,FN,FN1,FN4},{RN,FN,FN1,FN4,ON,ON1},{RN,FN,FN1,FN4,ON,ON2},{RN,FN,FN1,FN 4,ON,ON3}, {RN,FN,FN1,FN5}, {RN,FN,FN1,FN5,ON,ON1},{RN,FN,FN1,FN5,ON,ON2},{RN, FN,FN1,FN5,ON,ON3},{RN,FN,FN1,FN2,FN}, {RN,FN,FN1,FN2,FN4,ON,ON1},{RN,FN,FN1,FN2,FN4,ON,ON2},{RN,FN,FN1,FN2,FN4,ON,ON3},{RN,FN,FN1,FN2,FN5}, {RN,FN,FN1,FN2,FN5,ON,ON1},{RN,FN,FN1,FN2,FN5,ON,ON2},{RN,FN,FN1,FN2,FN5,ON,ON3},{RN,FN,FN1,FN3,FN4}, {RN,FN,FN1,FN3,FN4,ON,ON1},{RN,FN,FN1,FN3,FN4,ON,ON2},{RN,FN,FN1,FN3,FN4,ON,ON3},{RN,FN,FN1,FN3,FN5}, {RN,FN,FN1,FN3,FN5,ON,ON1},{RN,FN,FN1,FN3,FN5,ON,ON2}, {RN,FN,FN1,FN3,FN5,ON,ON3}

OR (“more-of”) Features

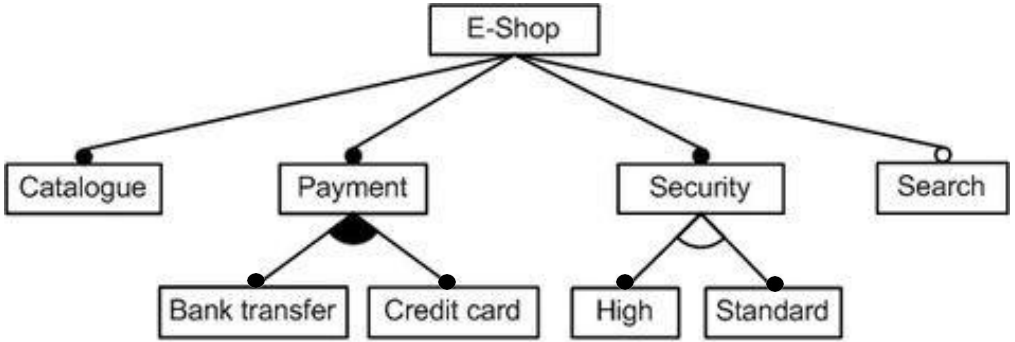
- The OR-feature, also called non-exclusive (“one-of”) choice is a set of features from which a non-empty set is selected to be included in the final description of the concept when the parent of the set is also included in the description.

OR features



- The feature diagram in the above figure shows a concept node RN that has two child features:
  - The first is a set of OR-features {MD1, MD2}
  - The second is the mandatory feature MS that has a set of OR-sub-features {MS1, MS2}
- The same condition is applied to the OR-sub-feature set {MS1, MS2} since it has a mandatory feature MS acting as a parent node
- The number of possible configurations for the feature diagram is 3 possible configurations for the OR-feature set {MD1, MD2} × 1 configuration for the mandatory feature MS × 3 possible configurations for the OR-sub-feature set {MS1, MS2}.
- The result is 9 possible configurations, which are: {RN,MD1,MS,MS1}, {RN,MD1,MS,MS2}, {RN,MD1,MS,MS1,MS2}, {RN,MD2,MS,MS1}, {RN,MD2,MS,MS2}, {RN,MD2,MS,MS1,MS2}, {RN,MD1,MD2,MS,MS1}, {RN,MD1,MD2,MS,MS2}, {RN,MD1,MD2,MS,MS1,MS2}.

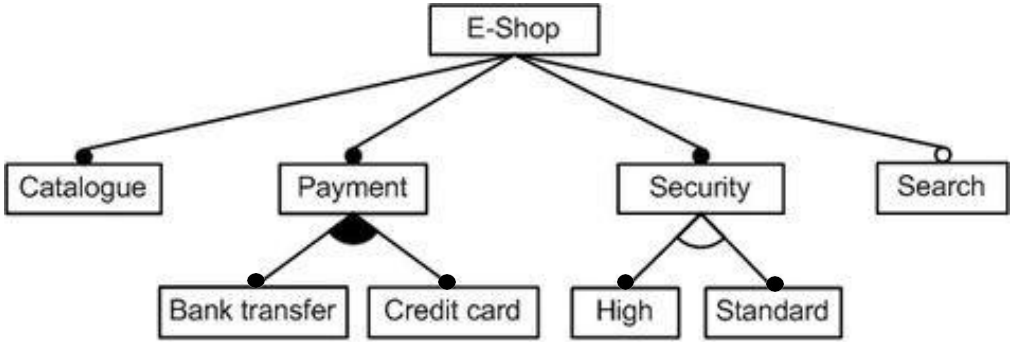
Example:



- The feature diagram in the above figure shows a concept node E-shop that has four child features:
  - The first is the mandatory feature- Catalogue
  - The second is the mandatory feature Payment that has a set of OR-sub-features {Bank transfer, Credit card}
  - The third is the mandatory feature Payment that has a set of alternate-sub-features {high, standard}
  - The four is the optional feature- Search
- The number of possible configurations for the feature diagram is:

  1 configuration for the mandatory feature {Catalogue} × 3 possible configurations for the OR-feature set {Bank transfer, Credit card} × 2 possible configurations for the alternate-sub-feature set {high, standard} × 2 configuration for the optional feature {Search} .

Example:



- Suppose Cat = catalogue, payment= pay, bank transfer= ban, credit card = cre, Security = sec, high = hig, standard = sta, search = sea
- The result is 12 possible configurations, which are:
  {cat, pay, ban, sec, hig}, {cat, pay, ban, sec, sta}, {cat, pay, cre, sec, hig}, {cat, pay, cre, sec, sta},
  {cat, pay, ban, cre, sec, hig}, {cat, pay, ban, cre, sec, sta},
  {cat, pay, ban, sec, hig, sea}, {cat, pay, ban, sec, sta , sea}, {cat, pay, cre, sec, hig , sea}, {cat, pay, cre, sec, sta , sea}, {cat, pay, ban, cre, sec, hig , sea}, {cat, pay, ban, cre, sec, sta , sea},

Component-Based Software Engineering

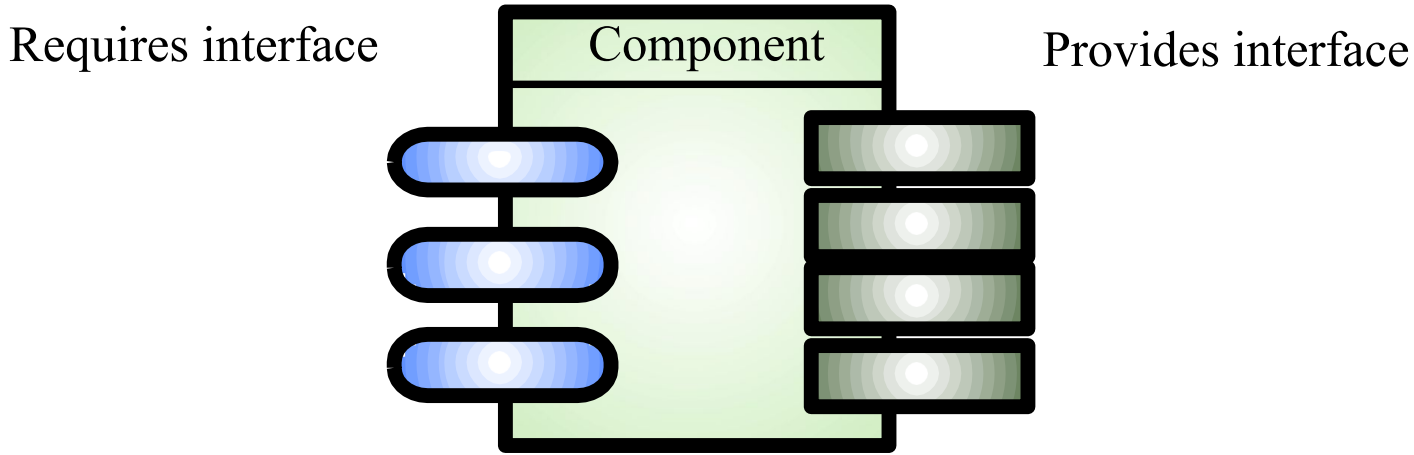
Component-based development

- Component-based software engineering (CBSE) is an approach to software development that relies on reuse
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- Components are more abstract than object classes and can be considered to be stand-alone service providers

Components

- Components provide a service without regard to where the component is executing or its programming language
  - A component is an independent executable entity that can be made up of one or more executable objects
  - The component interface is published and all interactions are through the published interface
- Components can range in size from simple functions to entire application systems

Component interfaces



Component interfaces

- Provides interface
  - Defines the services that are provided by the component to other components
- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components
- Component reusability
  - Should reflect stable domain abstractions
  - Should hide state representation
  - Should be as independent as possible
  - Should publish exceptions through the component interface
- There is a trade-off between reusability and usability.
  - The more general the interface, the greater the reusability but it is then more complex and hence less usable

Reusable components

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

Commercial Off-The-Shelf (COTS) Software

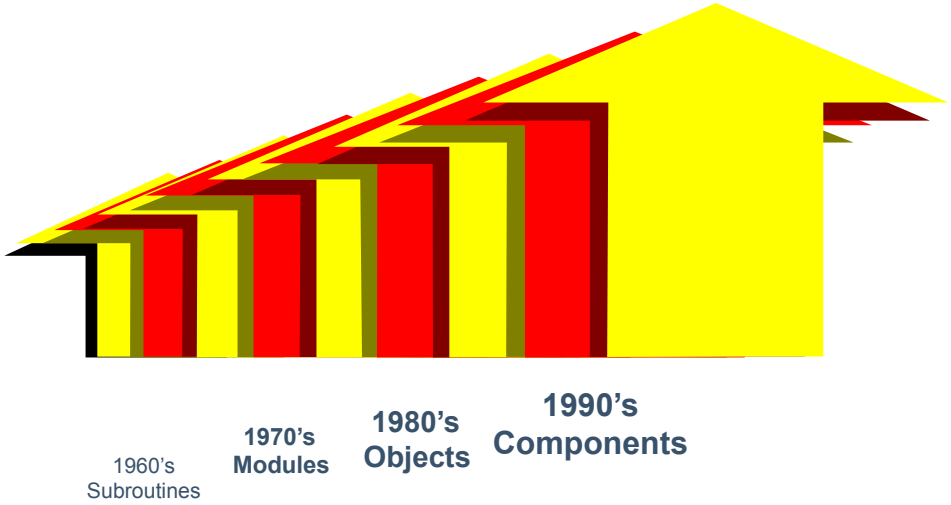
Commercial Off-The-Shelf (COTS) Software

- A commercial, pre-packaged software that can be configured (not altered at the code level) for use by various customers
- Examples: MS Office, Oracle ERP, Salesforce, etc.
- Even open-source software is sometimes used as COTS if it's used as-is without modifying the source

Types of COTS-Based Reuse

- COTS-Solution Systems
  - Single product (e.g., ERP system) configured to meet organization needs
  - Example: An ERP for a manufacturing company includes purchasing, logistics, and CRM modules—all integrated through a shared database and unified business rules
- COTS-Integrated Systems
  - Combines multiple COTS or legacy systems to create a complete application
  - Example: A procurement system integrating:
    - A legacy order processing system
    - A modern e-commerce platform
    - An email notification system

Reuse History



Focus was small-grained and opportunistic. Results fell short of expectations.

THANK YOU