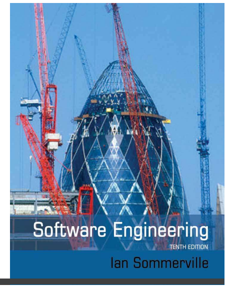
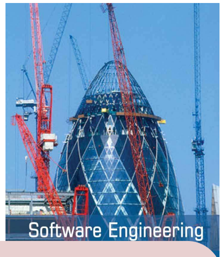
Software Reuse



Topics covered

THE REUSE LANDSCAPE

SOFTWARE PRODUCT LINES AND MORE…



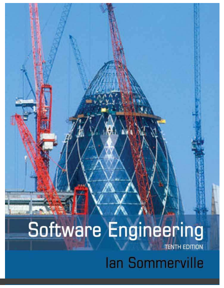
Software reuse

In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.

Software engineering has been more focused on original development, but it is now recognised that to achieve better software, more quickly and at lower cost, we need a design process that is based on systematic software reuse.

There has been a major switch to reuse-based development over the years.

Reuse-based software engineering



System/App reuse

- Complete systems/applications, may be reused.

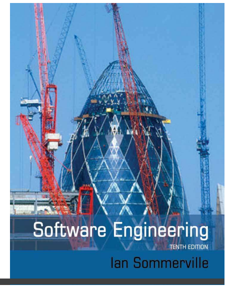
Component reuse

- Components of an application from sub-systems to single classes may be reused.

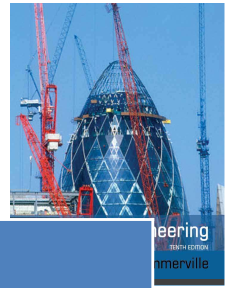
Class and function reuse

- Small-scale software components that implement a single well-defined class or function may be reused.

Benefits of software reuse



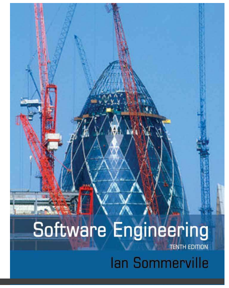
Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.



Benefits of software reuse

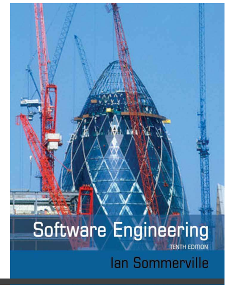
Benefit	Explanation
Lower development costs	• Development costs are proportional to the size of the software being developed. • Reusing software means that fewer lines of code have to be written.
Reduced process risk	• The cost of existing software is already known, whereas the costs of development are always a matter of judgment. • This is an important factor for project management because it reduces the margin of error in project cost estimation. • This is particularly true when relatively large software components such as subsystems are reused.
Standards compliance	• Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. • The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Problems with reuse



Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.

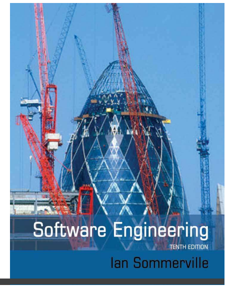
Problems with reuse



Problem	Explanation
Lack of tool/process support	• Some software tools do not support development with reuse. It may be difficult to integrate these tools with a component library system. • The software process assumed by these tools/process may not take reuse into account. • This is less problematic for object-oriented development tools.
Not-invented-here syndrome	• Some software engineers prefer to rewrite components because they believe they can improve on them. • This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

The reuse landscape

The reuse landscape

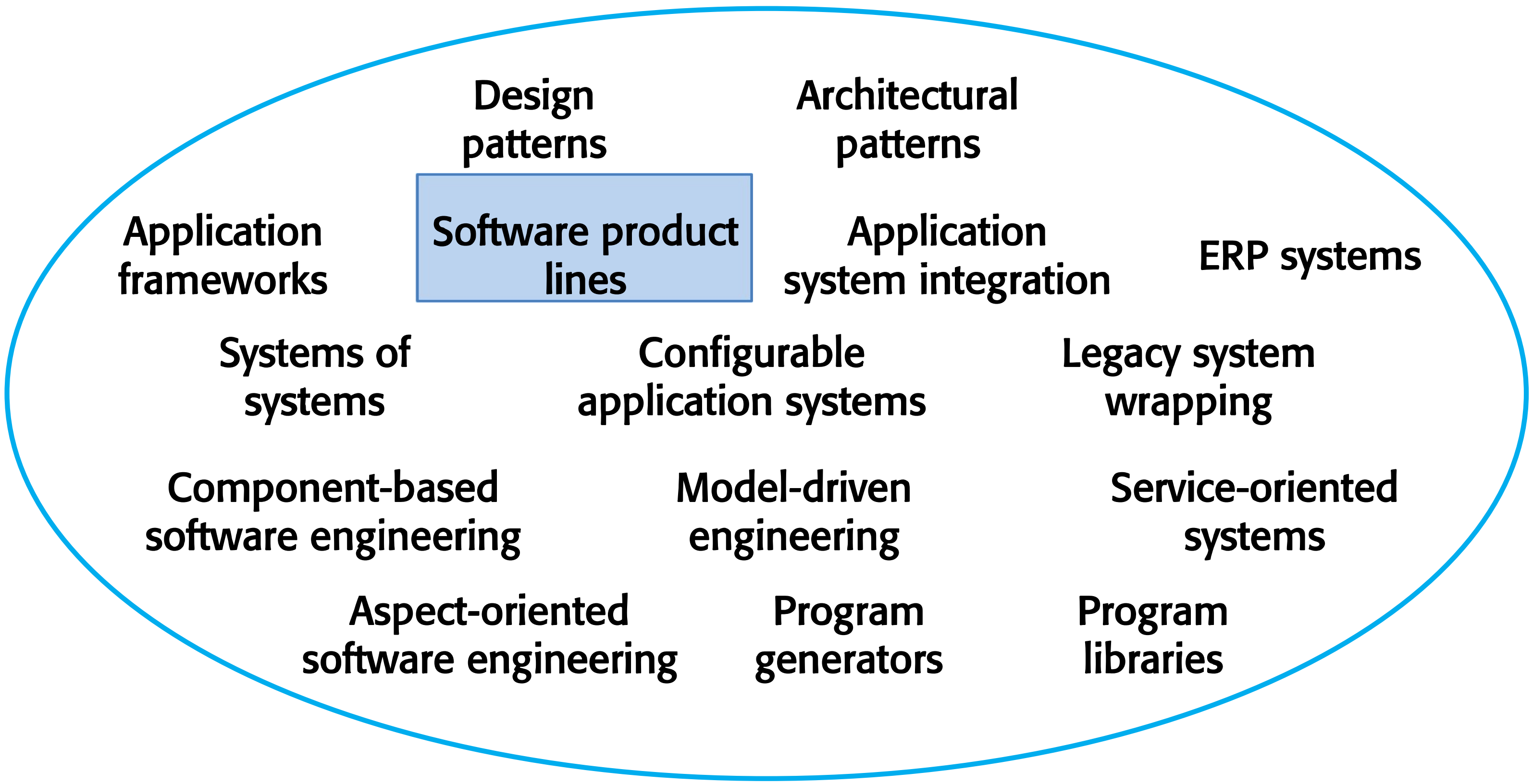


Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used.

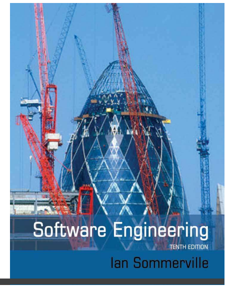
Reuse is possible at a range of levels from simple functions to complete application systems.

The reuse landscape covers the range of possible reuse techniques.

The reuse landscape

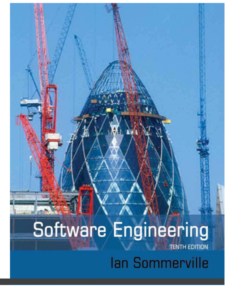


Approaches that support software reuse



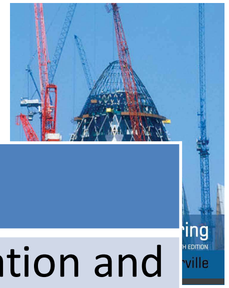
Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards.

Approaches that support software reuse



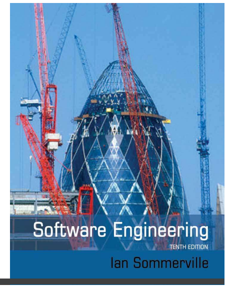
Approach	Description
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions.
ERP (Enterprise Resource Planning) systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization. ERP is a category of business-management software—typically a suite of integrated applications—that an organization can use to collect, store, manage and interpret data from many business activities. ERP is a business process management software that manages and integrates a company's finance, supply chain, operations, reporting, manufacturing and human resources activities.
Legacy system wrapping	Legacy systems are 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models and code is generated from these models.

Approaches that support software reuse

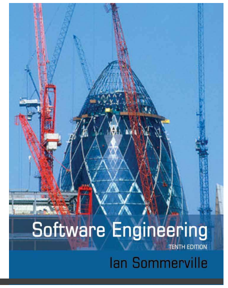


Approach	Description
Program generators	A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model. In practice, generators are typically compilers for domain-specific languages (DSLs). A domain-specific language is a special-purpose programming language for a particular software domain.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. In SOA (Service-oriented architecture), services use protocols that describe how they pass and parse messages using description metadata. This metadata describes both the functional characteristics of the service and quality-of-service characteristics. SOA aims to allow users to combine large chunks of functionality to form applications which are built purely from existing services and combining them in an ad hoc manner.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	A SoS brings together a set of systems for a task that none of the systems can accomplish on its own. Each constituent system keeps its own management, goals, and resources while coordinating within the SoS and adapting to meet SoS goals. E.g. Air traffic

Reuse planning factors

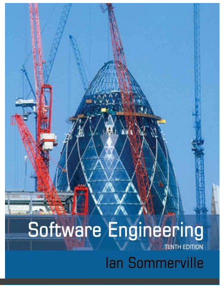


- THE DEVELOPMENT SCHEDULE FOR THE SOFTWARE.
- THE EXPECTED SOFTWARE LIFETIME.
- THE BACKGROUND, SKILLS AND EXPERIENCE OF THE DEVELOPMENT TEAM.
- THE CRITICALITY OF THE SOFTWARE AND ITS NFRS.
- THE APPLICATION DOMAIN.
- THE EXECUTION PLATFORM FOR THE SOFTWARE.



Software Product Lines

SPL: Definition



A *software product line* is a set of software-intensive systems that share a common, managed set of features

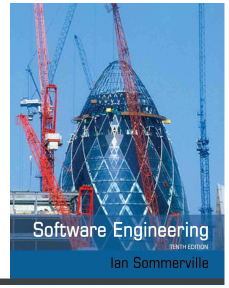
satisfying the specific needs of a particular market segment or mission and

are developed from a common set of core assets in a prescribed way.

Software product line practice is the systematic use of core assets to assemble, instantiate, or generate the multiple products that constitute a software product line.

Software product line practice involves strategic, large-grained reuse.

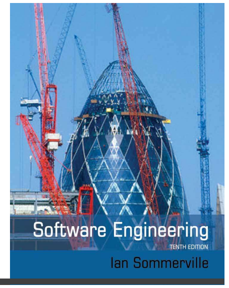
Core assets

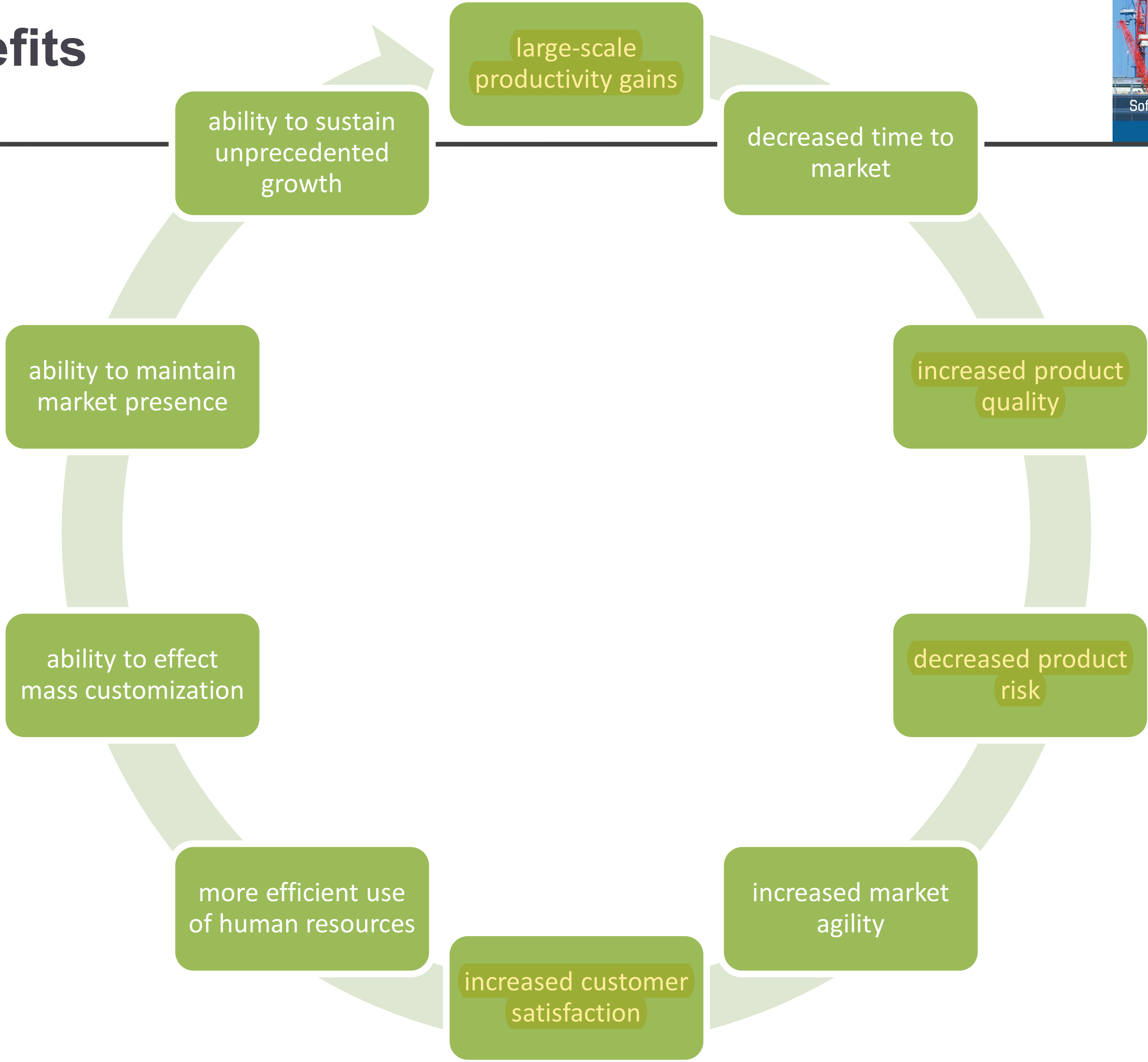


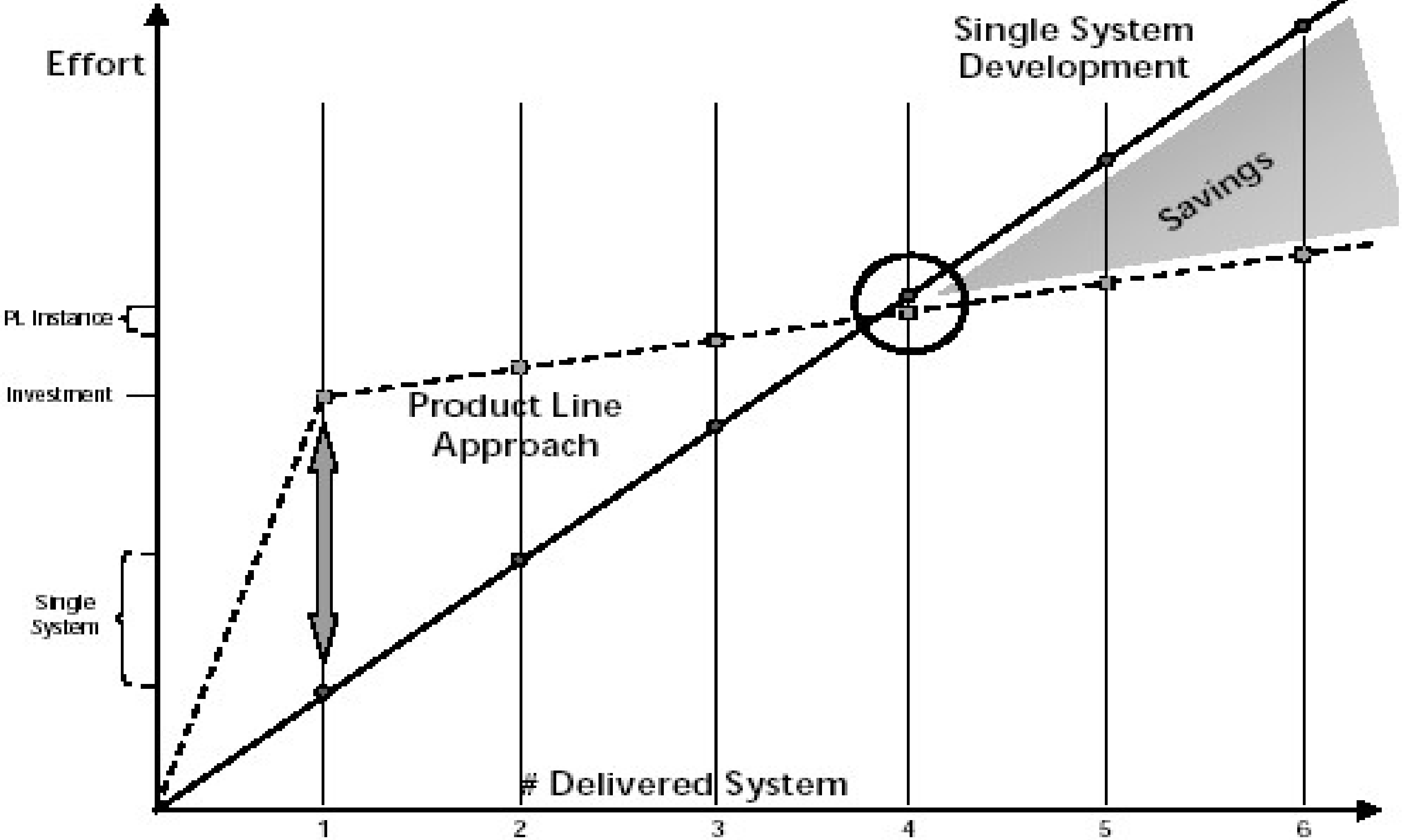
Core assets are those reusable artifacts and resources that form the basis for the software product line.

Core assets often include, but are not limited to, the **architecture, reusable software components, domain models, requirements statements, documentation, specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions**.

Benefits

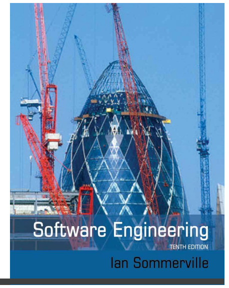

Software Engineering
Ian Sommerville


large-scale productivity gains
decreased time to market
increased product quality
decreased product risk
increased market agility
increased customer satisfaction
more efficient use of human resources
ability to effect mass customization
ability to maintain market presence
ability to sustain unprecedented growth



Effort
Single System Development
Savings
PL Instance
Investment
Product Line Approach
Single System
Delivered System
1
2
3
4
5
6

Domain

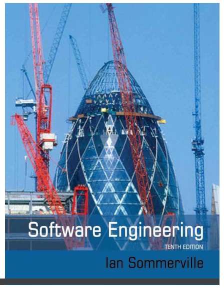


A *domain* is a specialized body of knowledge, an area of expertise, or a collection of related functionality.

For example, the telecommunications domain is a set of telecommunications functionalities

The *product family* is that set of products we call the product line.

How is production made more economical?



Each product is formed by

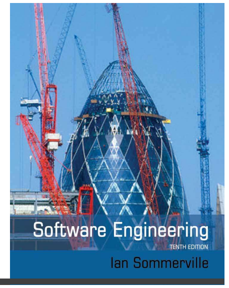
- taking applicable components from the base of common assets,
- tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance,
- adding any new components that may be necessary,
- assembling the collection according to the rules of a product-line-wide architecture.

Building a new product (system) becomes more a matter of assembly or generation than one of creation;

- the predominant activity is integration rather than programming.

For each software product line, there is a predefined guide or plan that specifies the exact product-building approach.

Other concepts

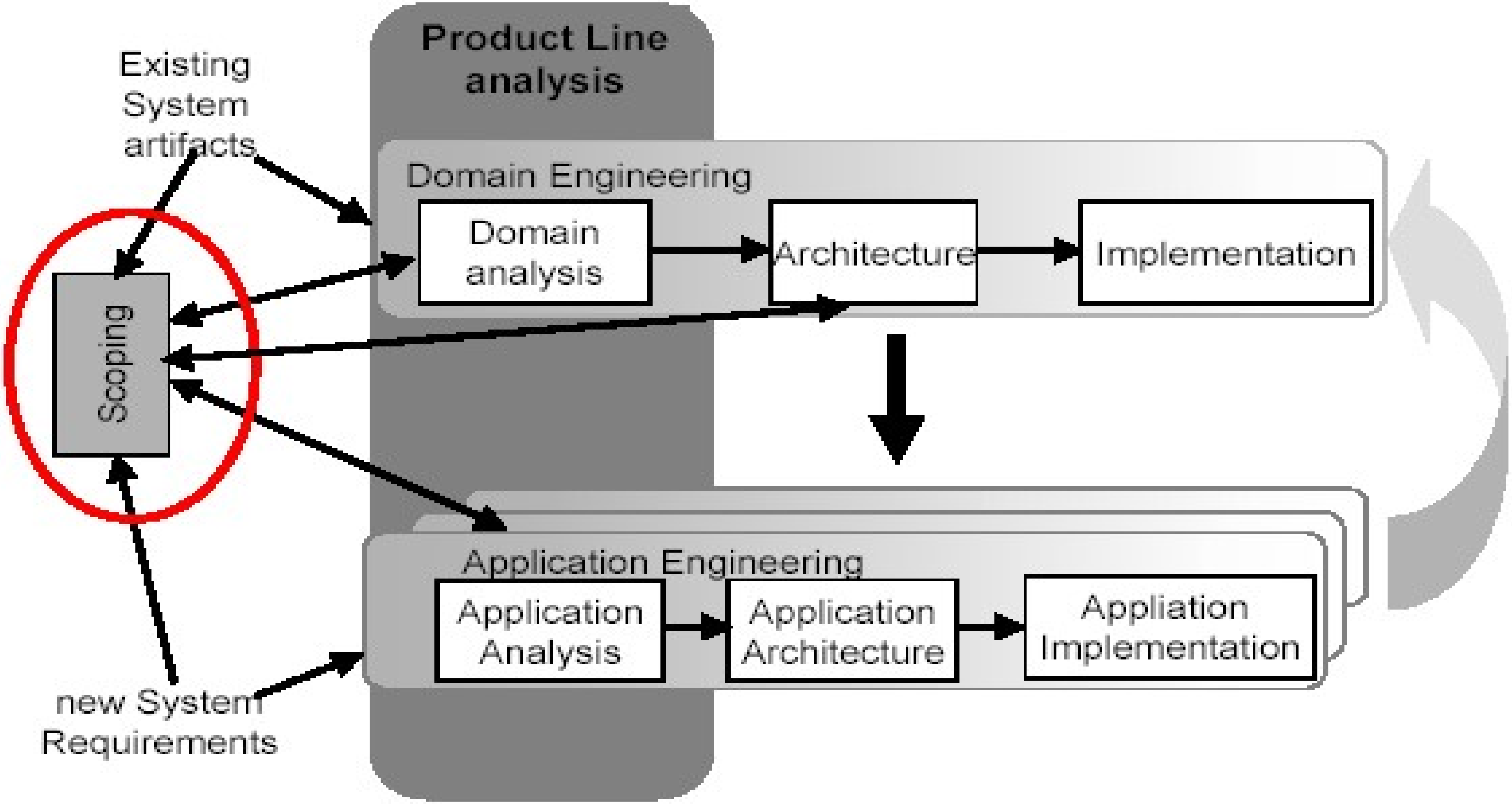


The software assets in the core asset base are sometimes called a *platform*.

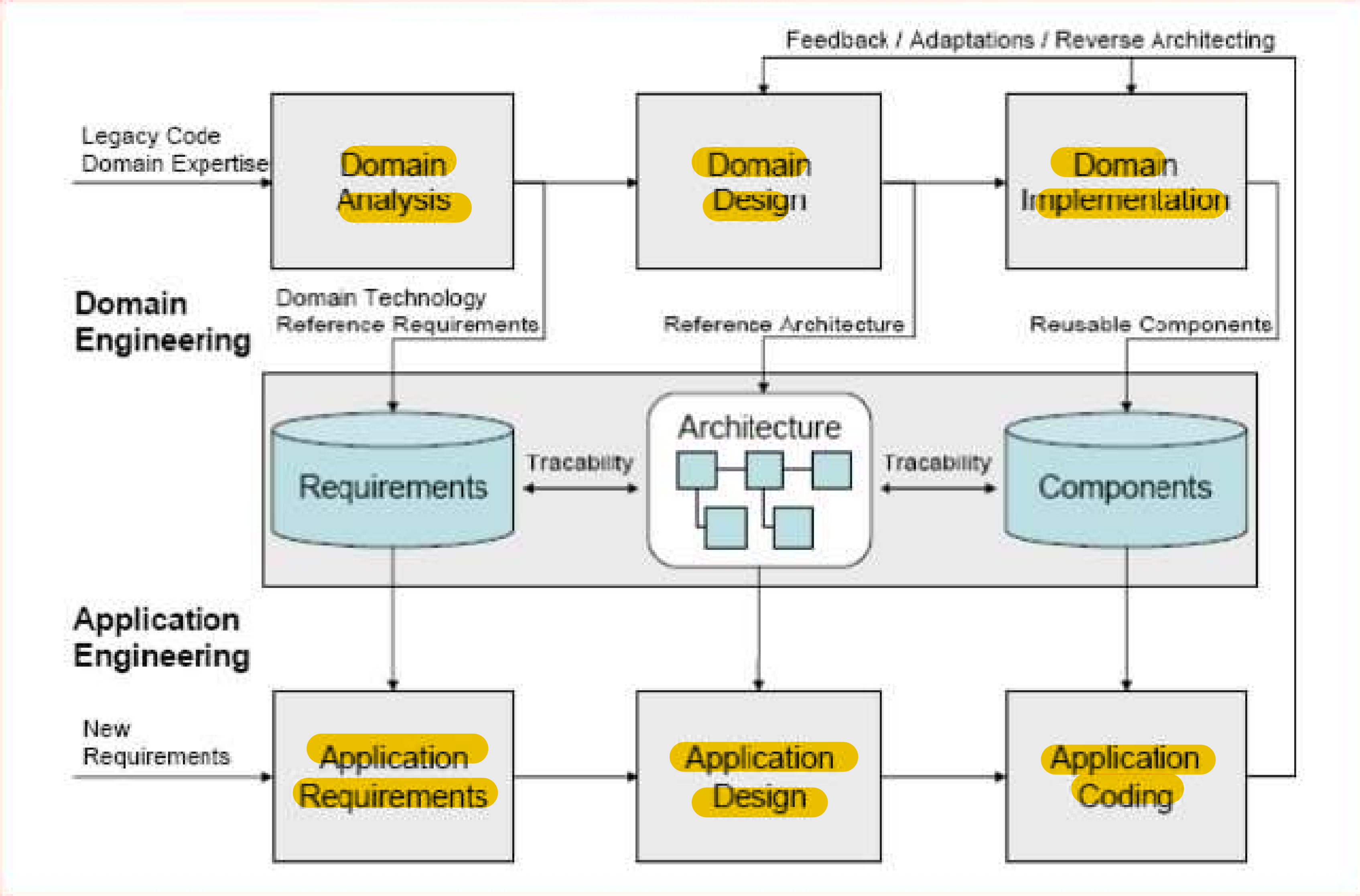
What we call core asset development is sometimes referred to as *domain engineering*.

What we call product development is sometimes referred to as *application engineering*.

SPL processes

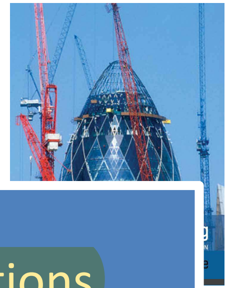


SPL Process and Organization



SPL activities

Requirements engineering for product lines



Requirements elicitation for a product line must capture anticipated variations explicitly over the foreseeable lifetime of the product line.

This means that the community of stakeholders is probably larger than for single-system requirements elicitation

it may well include domain experts, market experts, and others.

Requirements elicitation focuses on:

the scope, explicitly capturing the anticipated variation by the application of domain analysis techniques,	the incorporation of existing domain analysis models,	capturing the variations that are expected to occur over the lifetime of the PL

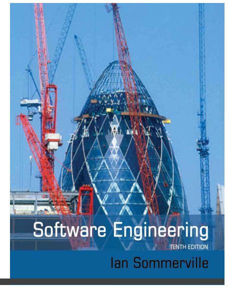
RE for SPL

Requirements analysis for a product line involves finding commonalities and identifying variations.

Requirements analysis includes a commonality and variability analysis on the elicited product line requirements to identify the opportunities for large-grained reuse within the product line.

Two such techniques are **F**eature-**O**riented **D**omain **A**nalysis (….) and use cases

Domain analysis techniques



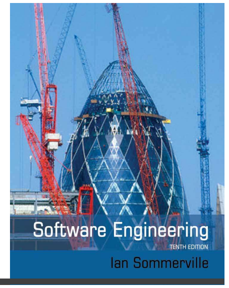
These techniques can be used:

- to expand the scope of the requirements elicitation,
- to identify and plan for anticipated changes,
- to determine fundamental commonalities and variations in the products of the SPL
- to support the creation of robust architectures.

Feature modeling facilitates the identification and analysis of the product line's commonality and variability and provides a natural vehicle for requirements specification.

Other techniques: Use case modeling

Feature modeling

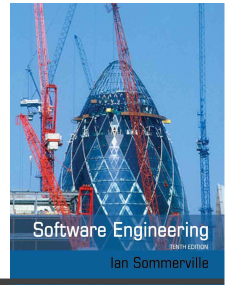


This technique can be used to complement object and use case modeling and to organize the results of the commonality and variability analysis in preparation for reuse.

Features are user-visible aspects or characteristics of a system that are organized into a tree of And/Or nodes to identify the commonalities and variabilities within the system.

The commonalities and variabilities within those features are then exploited to create a set of reference models (that is, software architectures and components) that can be used to implement the products of that family.

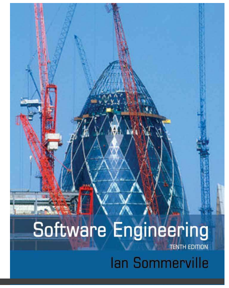
Features and feature model

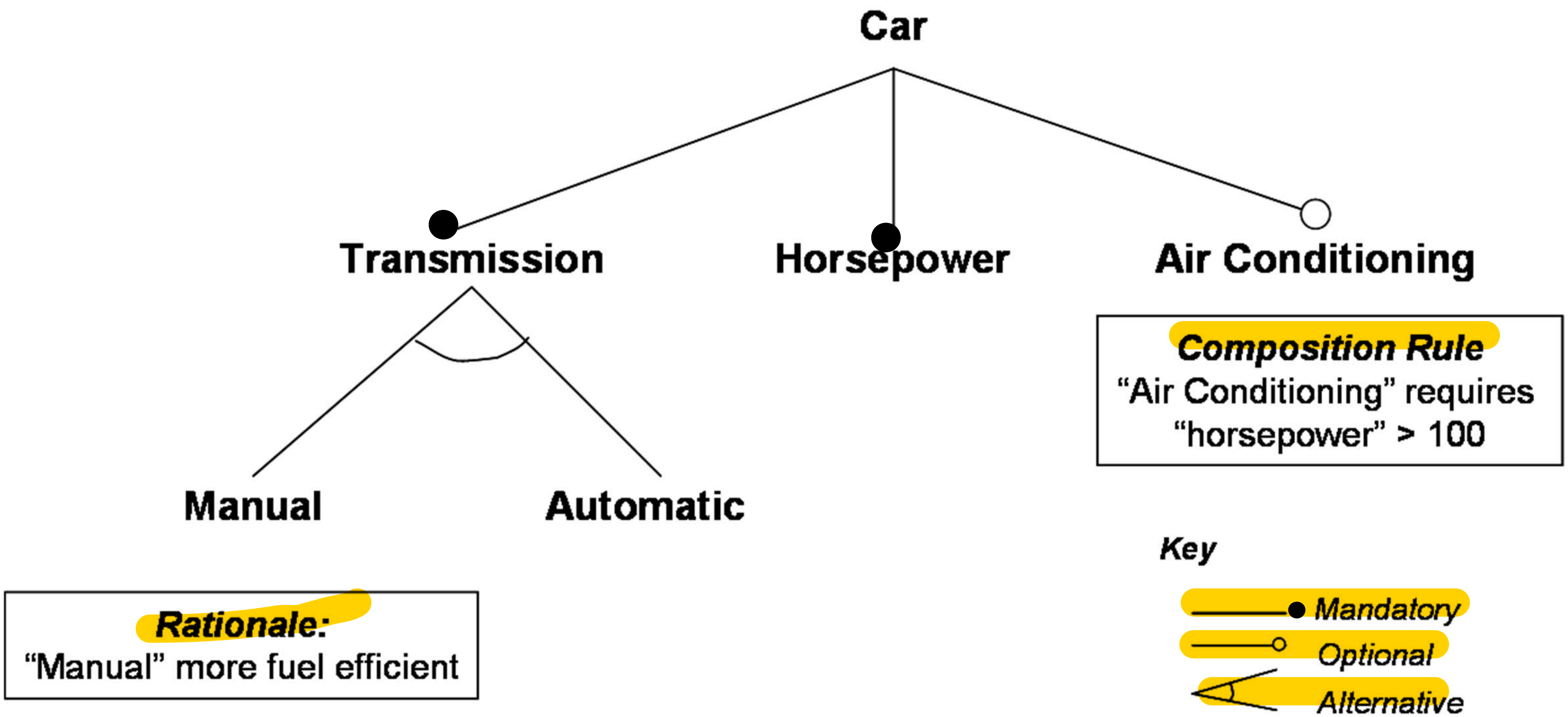


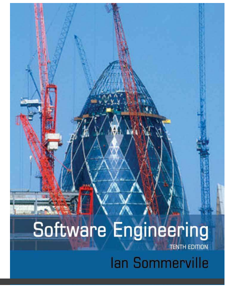
A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a product line.

A *feature model* consists of one or more *feature diagrams*, which organize features into hierarchies.

Feature diagram of a car







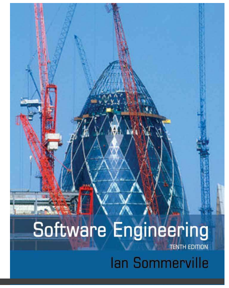
Feature-Oriented Domain Analysis

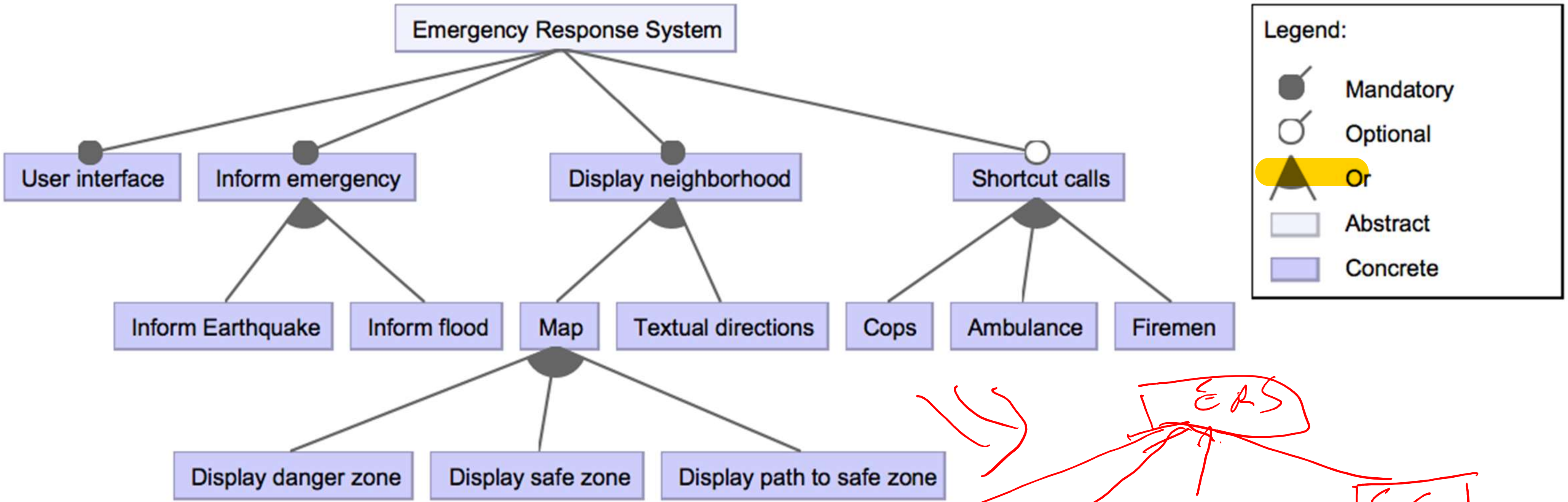
Feature diagram. The diagram depicts a hierarchical decomposition of features with mandatory, alternative and optional relationships.

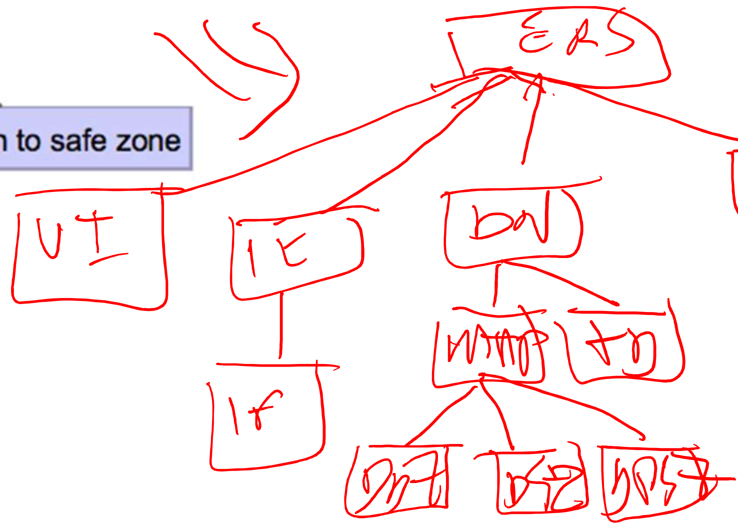
Feature definitions. Description of all features

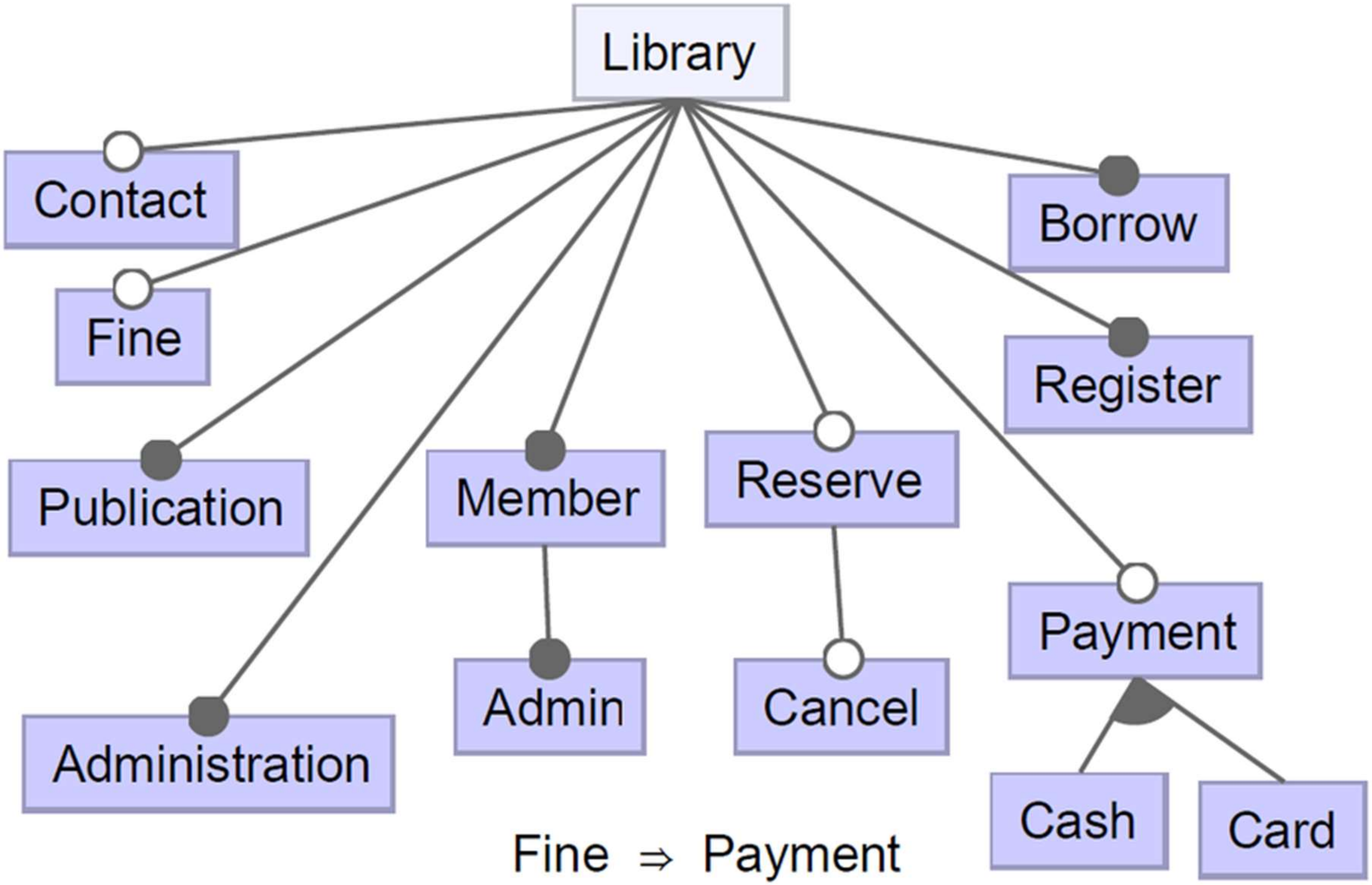
Composition rules. These rules indicate which feature combinations are valid and which are not.

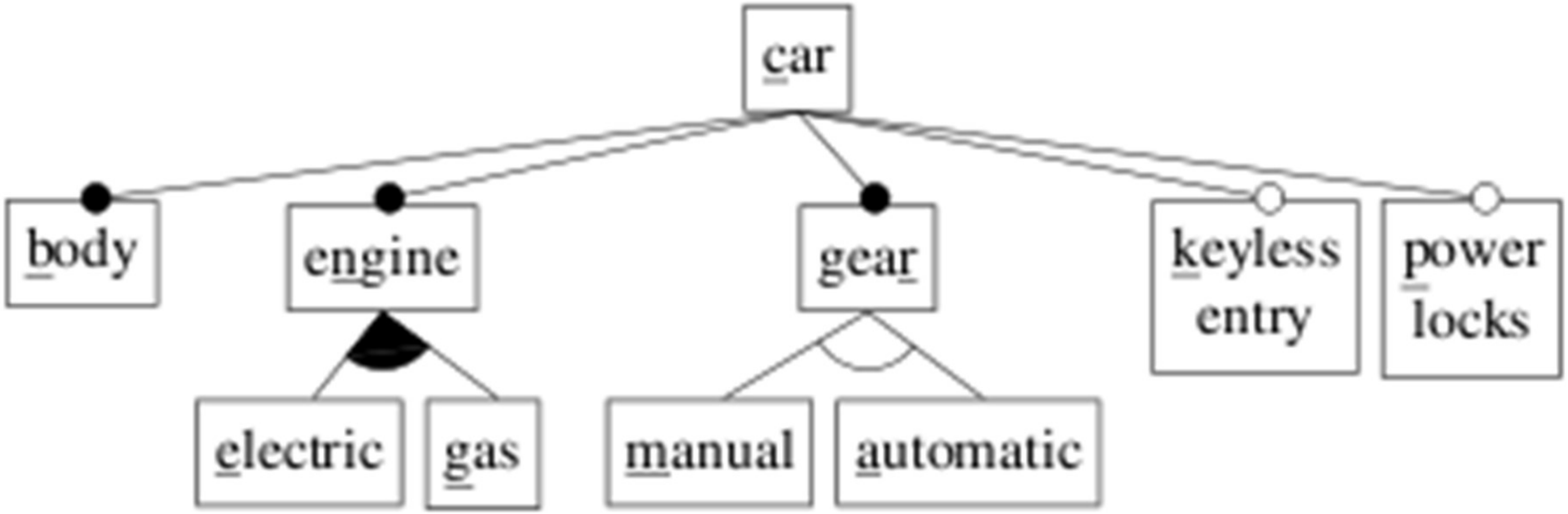
Rationale for features. The rationale for choosing or not choosing a particular feature, indicating the trade-offs.


Software Engineering
Ian Sommerville

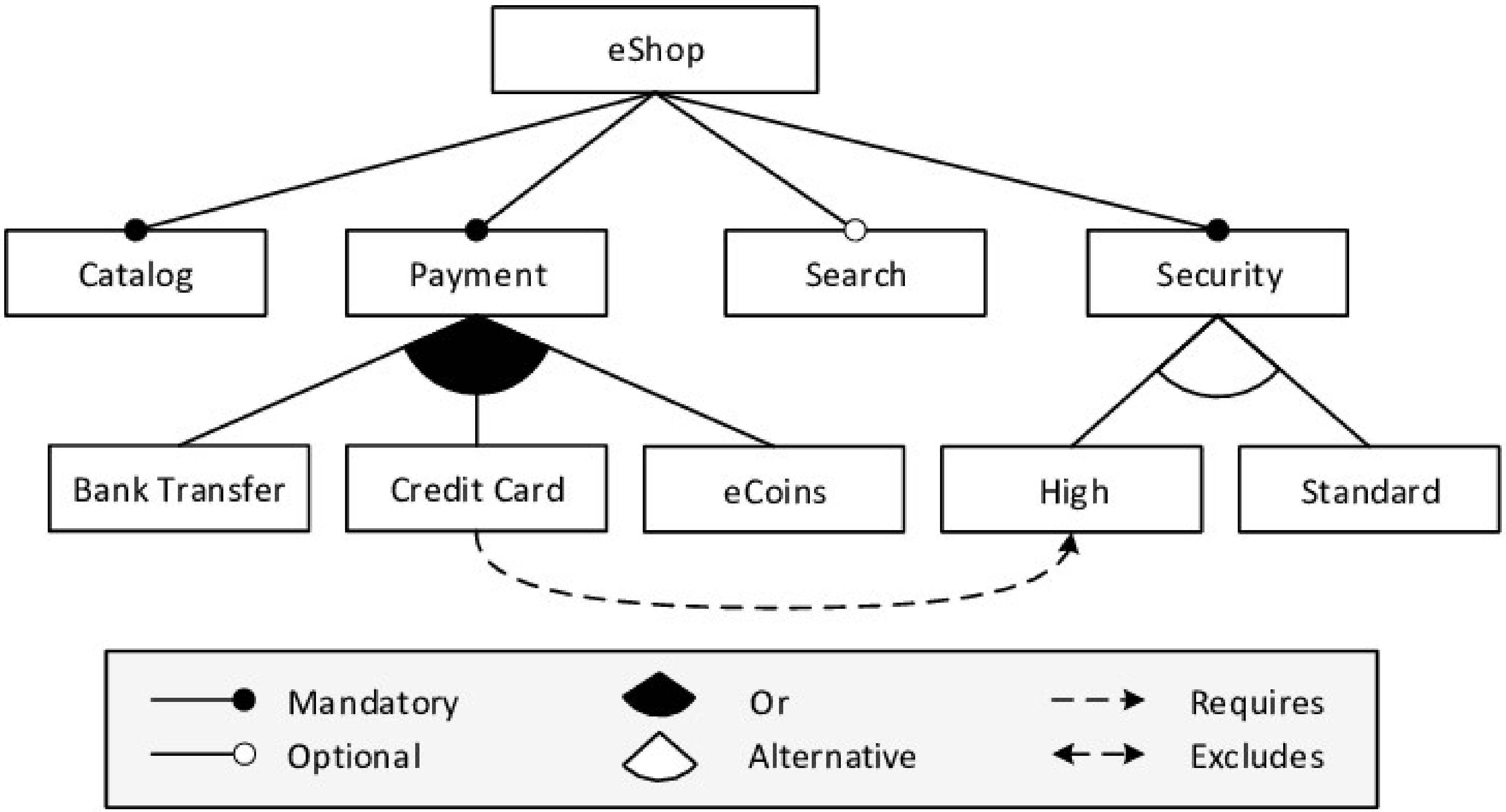

Emergency Response System
User interface
Inform emergency
Display neighborhood
Shortcut calls
Inform Earthquake
Inform flood
Map
Textual directions
Cops
Ambulance
Firemen
Display danger zone
Display safe zone
Display path to safe zone
Legend:
Mandatory
Optional
Or
Abstract
Concrete


ERS
UI
IE
DN
SC

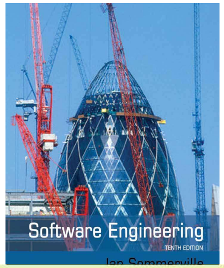

Library
Contact
Fine
Publication
Administration
Member
Admin
Reserve
Cancel
Borrow
Register
Payment
Cash
Card
Fine ⇒ Payment

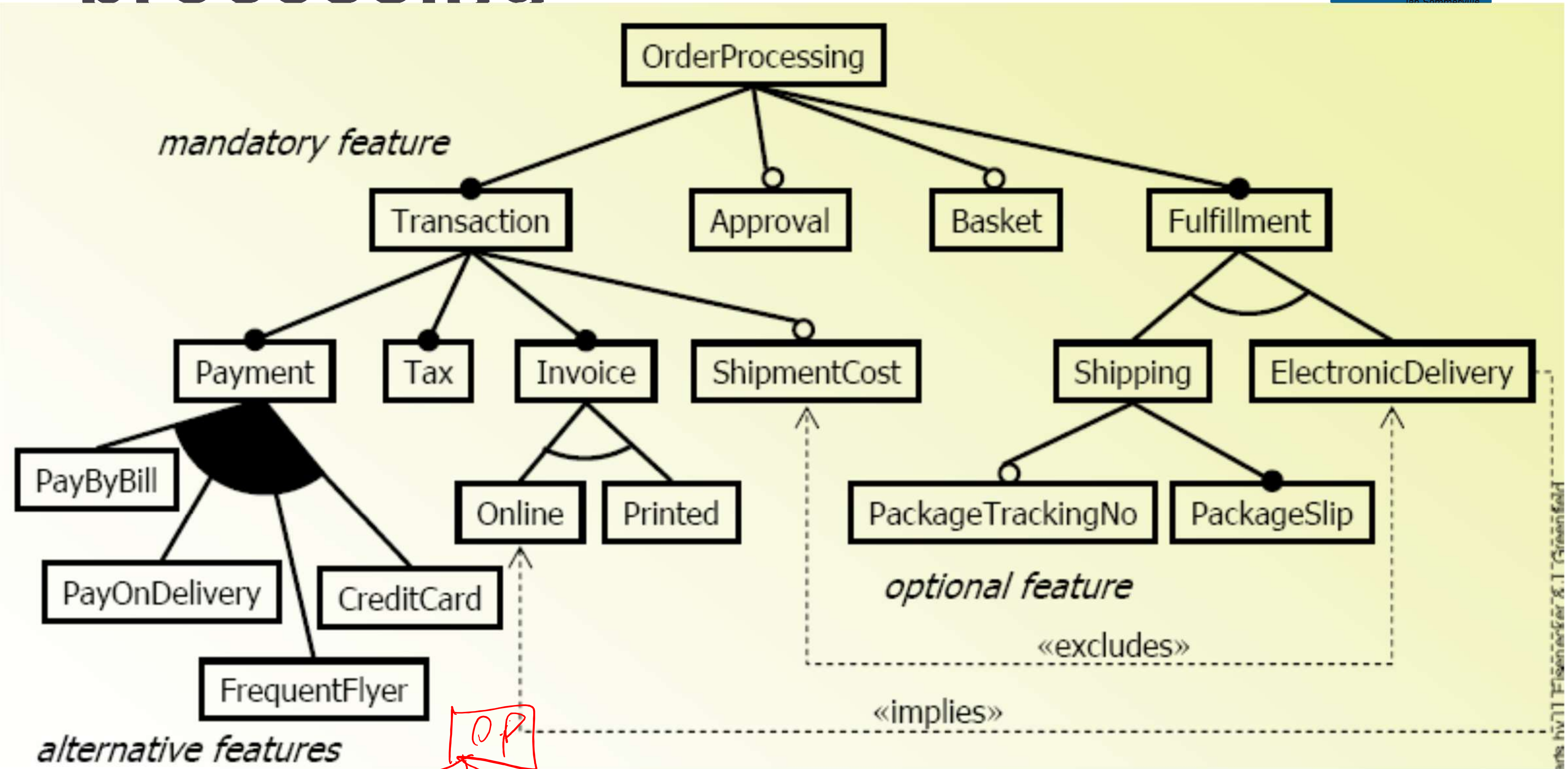


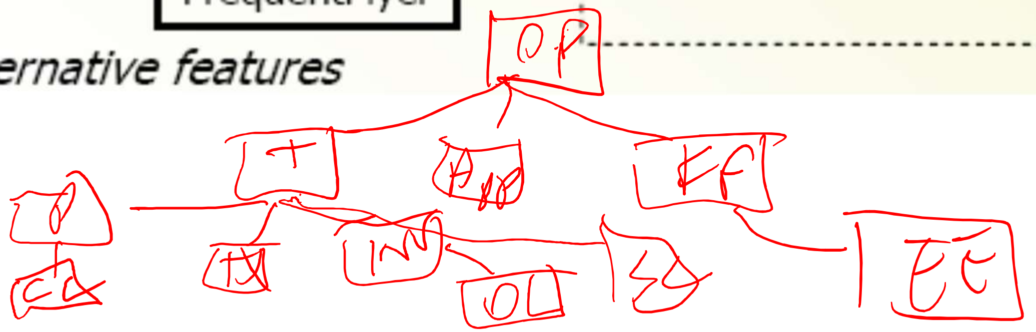
keyless_entry → power_locks


eShop
Catalog
Payment
Search
Security
Bank Transfer
Credit Card
eCoins
High
Standard
Mandatory
Optional
Or
Alternative
Requires
Excludes

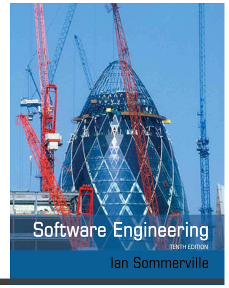
Feature model for order processing







UC Modeling with the PLUS approach [H.Gomaa]



Feature modeling

Use case modeling

<<kernel>>, <<optional>>, <<alternative>>

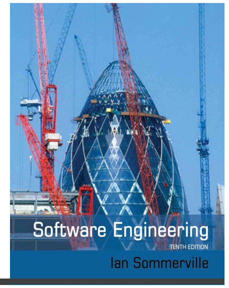
Static modeling

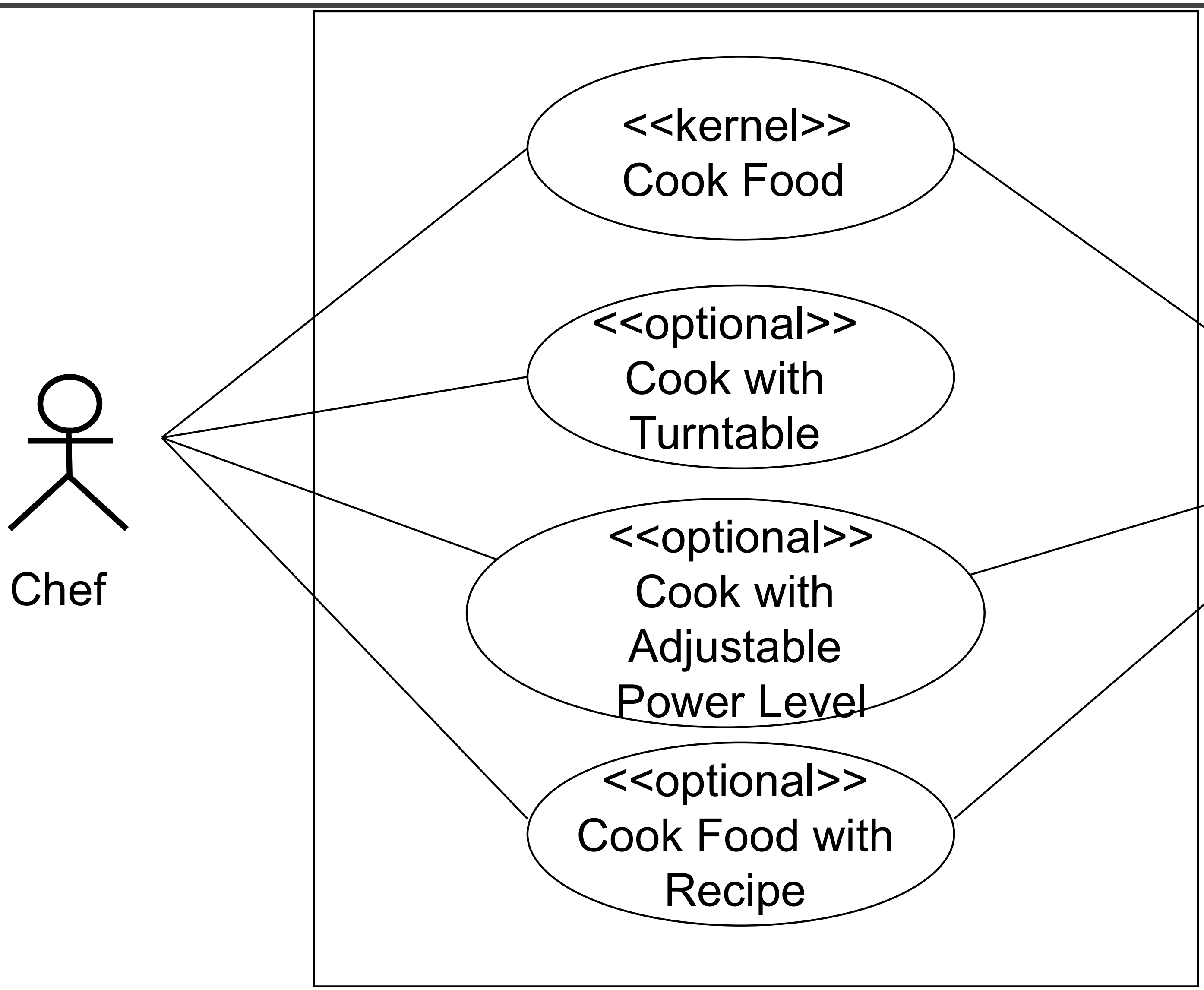
<<kernel>>, <<optional>>, <<alternative>> in UML class diagrams

PLUS: UC modeling

Kernel UC	UC that are required by all members of the PL
Optional UC	They are required by some, but not all the UC in the PL
Alternative UC	Different versions of the UC are required by different PL members They are usually mutually exclusive

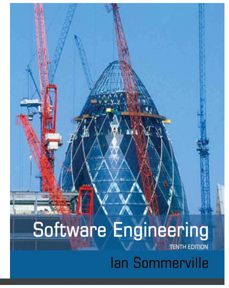
PLUS Example: Use Cases

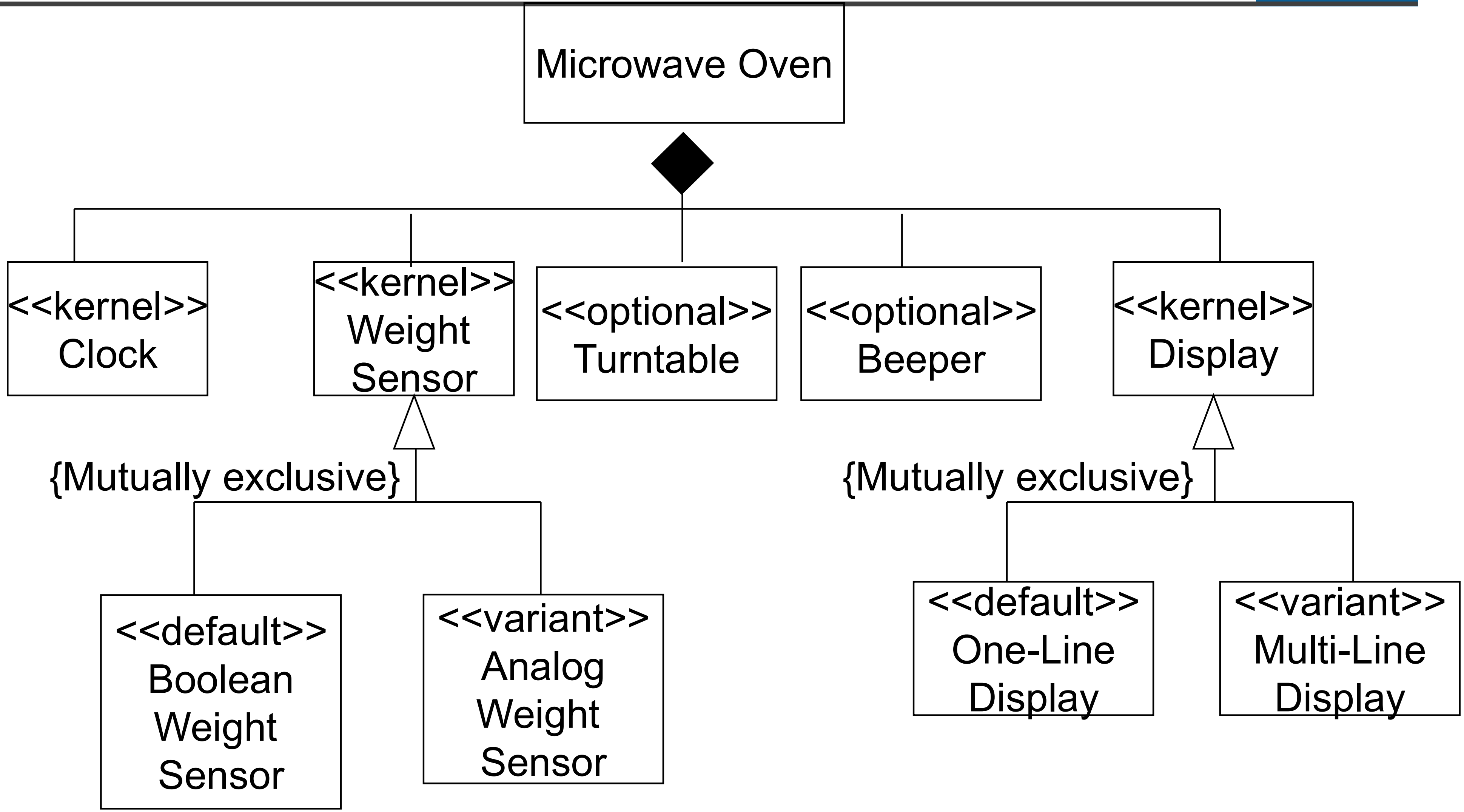




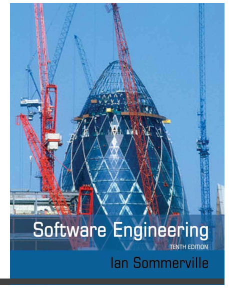
Adapted from [Gomaa05]

PLUS : Static Modeling





Adapted from [Gomaa05]



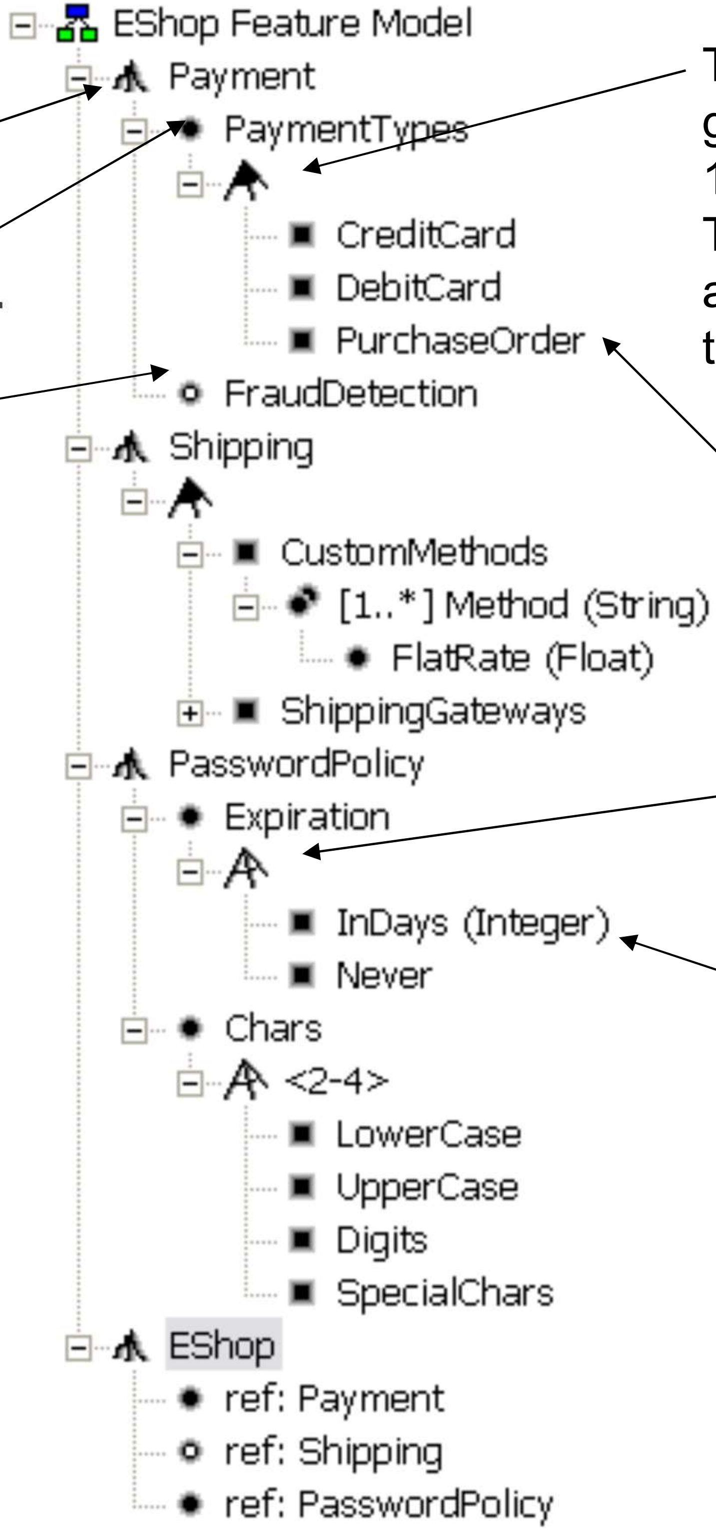
FeaturePlugin

- **Feature Modeling Plug-In for Eclipse**
- The tool supports
  - cardinality-based feature modeling
  - specialization of feature diagrams
  - configuration based on feature diagrams
- http://www.swen.uwaterloo.ca/~kczarnec/

root symbol

obligatory

optional

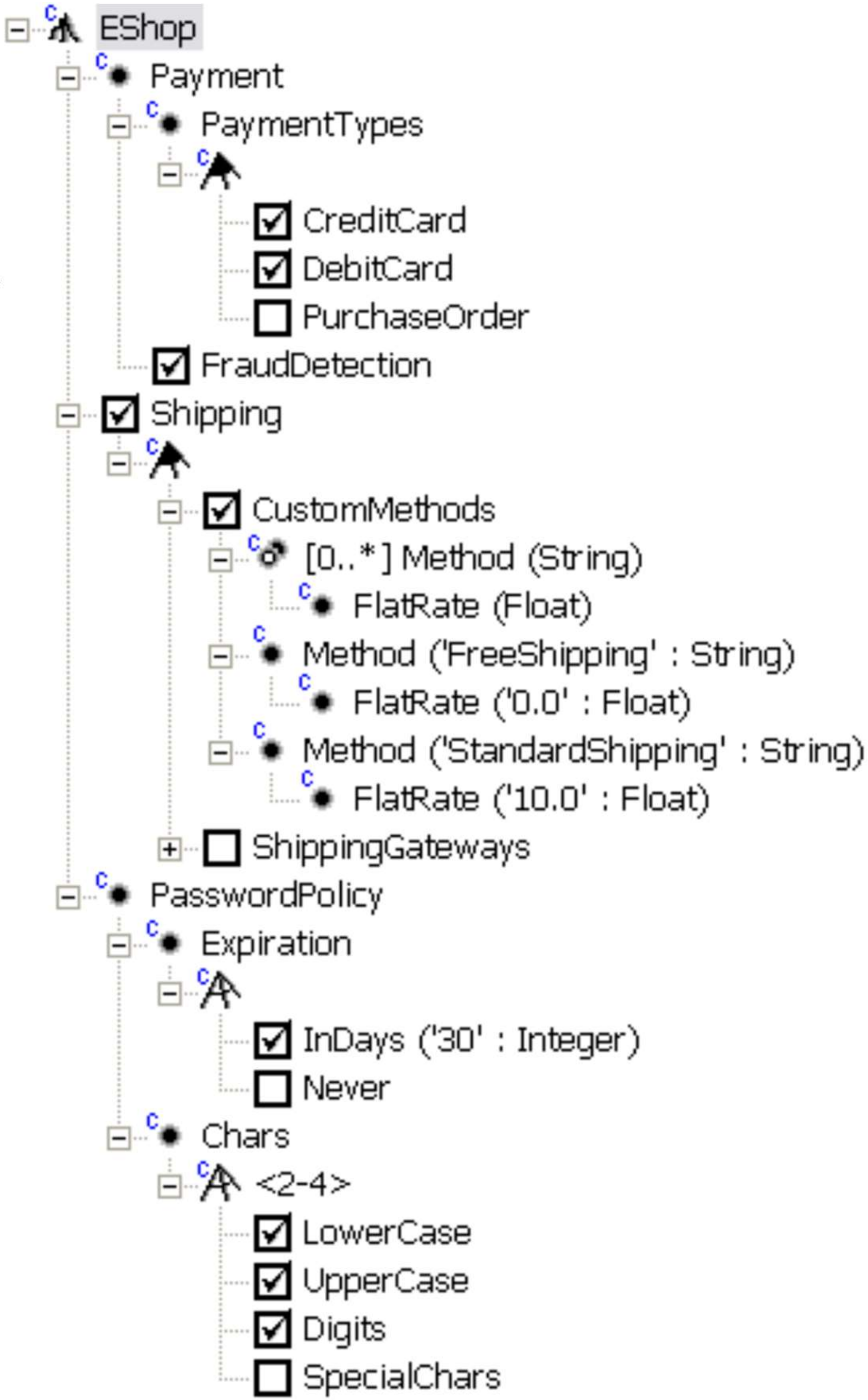


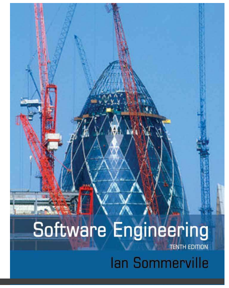
The **group symbol** indicates group cardinality 1– *k*, where *k* is the group size. Thus, our shop can support any non-empty subset of the three payment types.

Grouped features

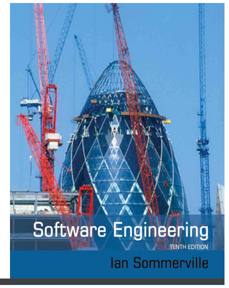
group symbol indicates group cardinality of 1– 1

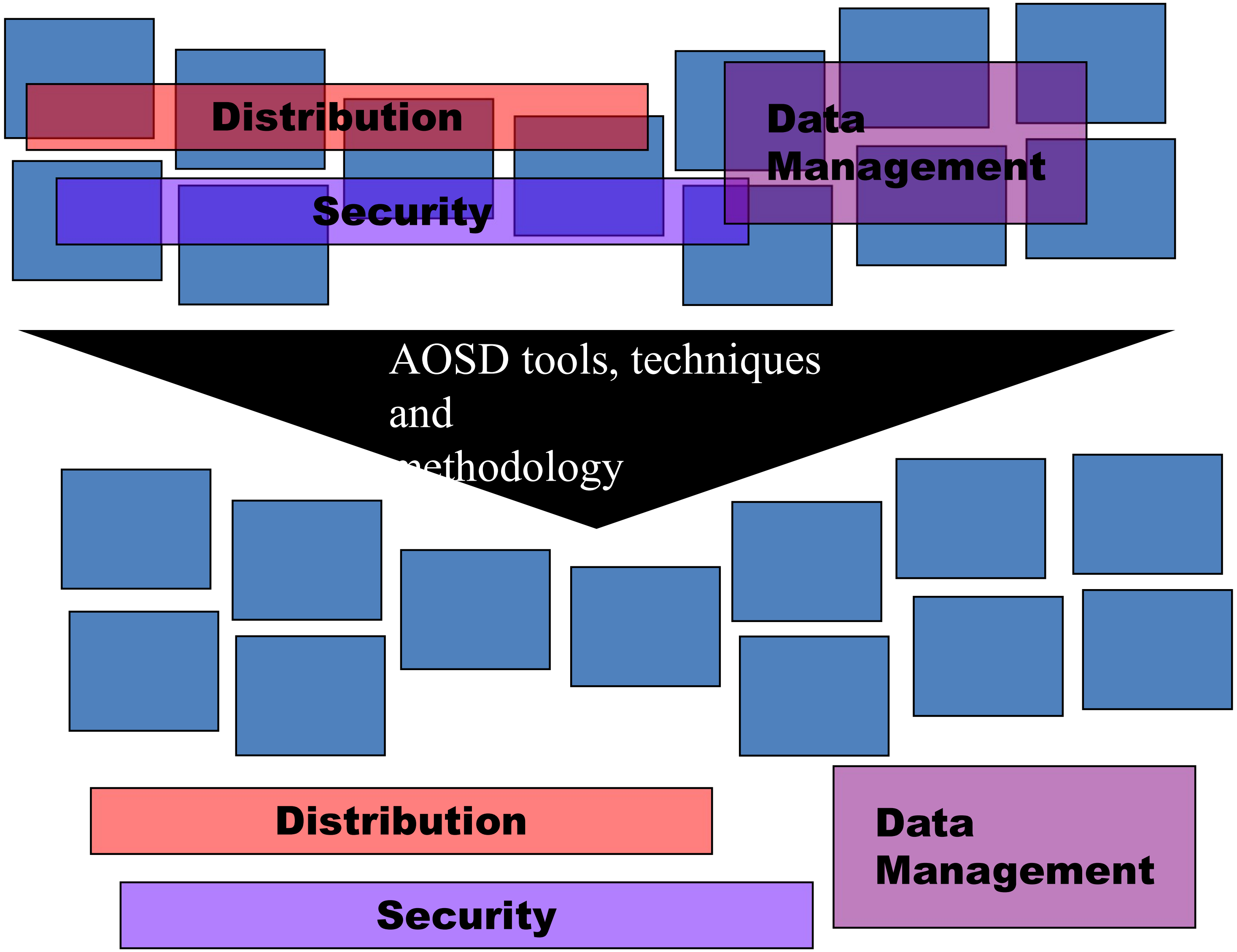
A feature can have an *attribute*, its type is indicated in parenthesis


EShop
Payment
PaymentTypes
CreditCard
DebitCard
PurchaseOrder
FraudDetection
Shipping
CustomMethods
[0..*] Method (String)
FlatRate (Float)
Method ('FreeShipping' : String)
FlatRate ('0.0' : Float)
Method ('StandardShipping' : String)
FlatRate ('10.0' : Float)
ShippingGateways
PasswordPolicy
Expiration
InDays ('30' : Integer)
Never
Chars
<2-4>
LowerCase
UpperCase
Digits
SpecialChars

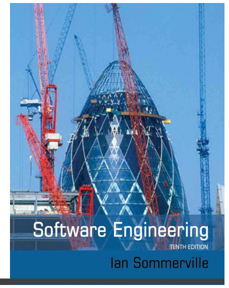

Software Engineering
Ian Sommerville

Aspect-Oriented Software Development (AOSD)





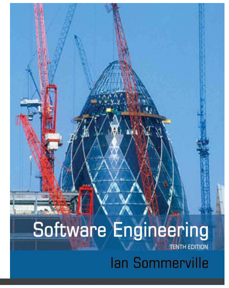
A Definition of AOSD



- **AOSD**: systematic *identification*, *modularisation*, *representation* and *composition* of crosscutting concerns [1]

[1] Rashid, A., Moreira, A., Araújo, J. “Modularisation and Composition of Aspectual Requirements”, Proceedings of 2nd International Conference on Aspect-Oriented Software Development, ACM, 2003.

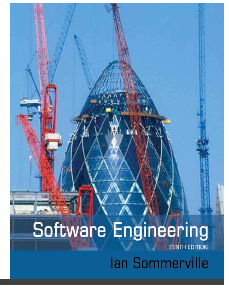
The Problem of Crosscutting Concerns

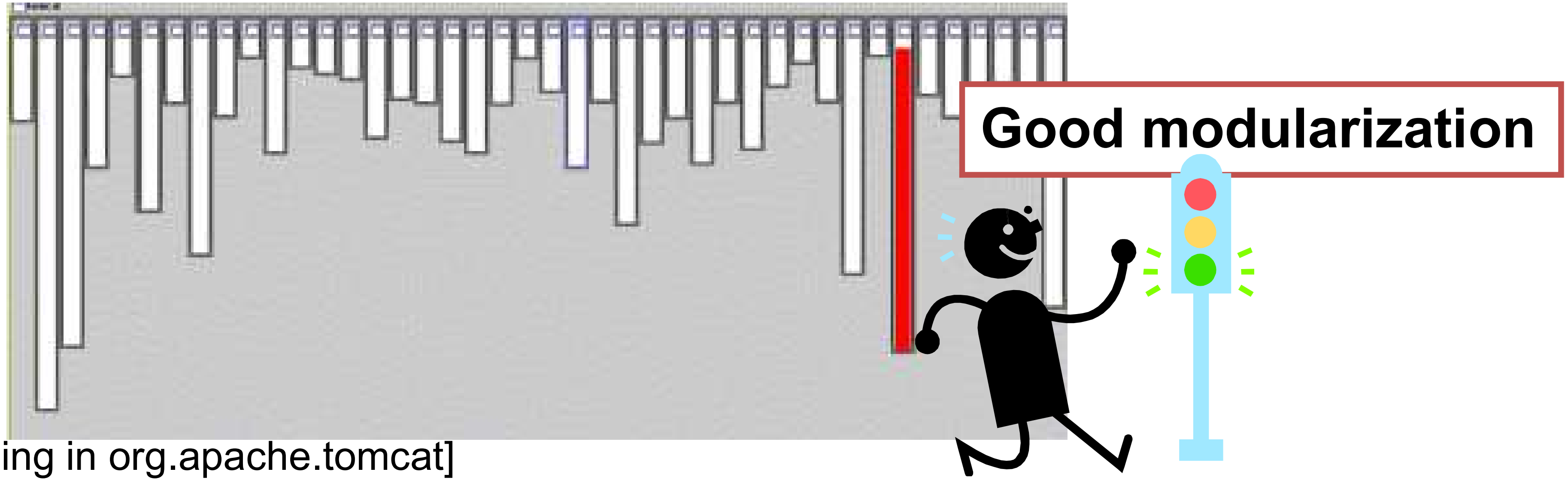


✧ Broadly-scoped concerns

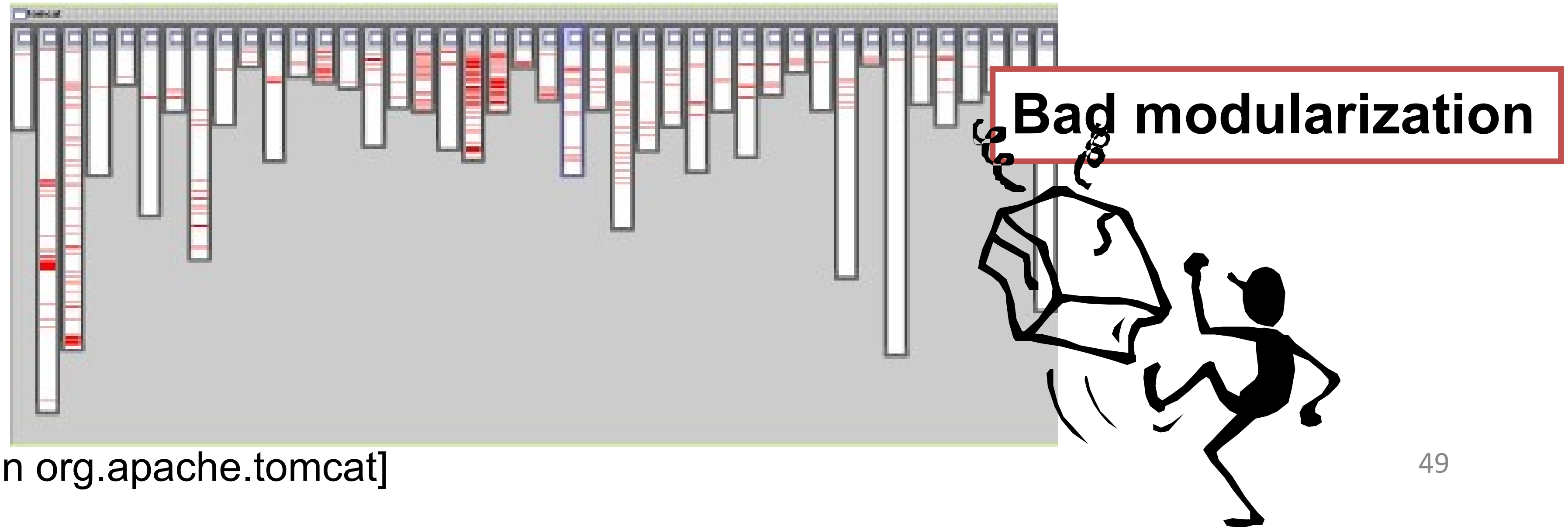
- Distribution, security, real-time constraints, etc.
- Crosscutting in nature
- Severely constrain quality attributes and separation of concerns

Crosscutting Concerns Affect Modularization



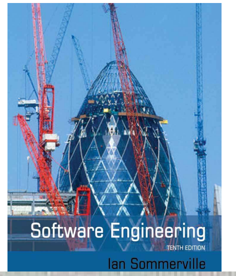


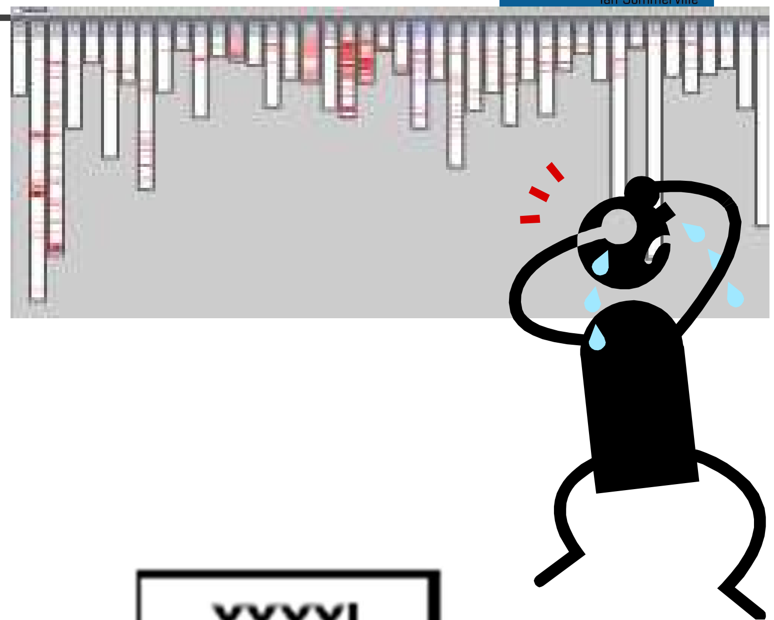
[XML parsing in org.apache.tomcat]



[logging in org.apache.tomcat]

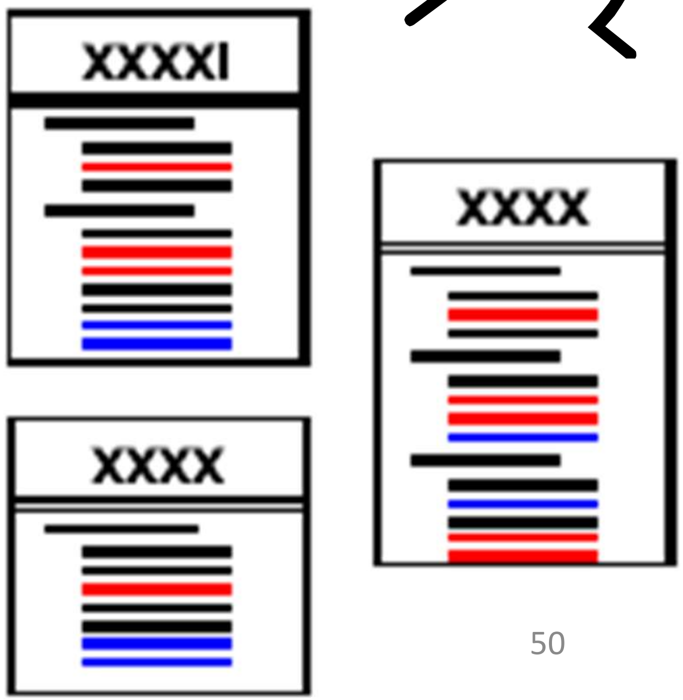
Resulting Problems



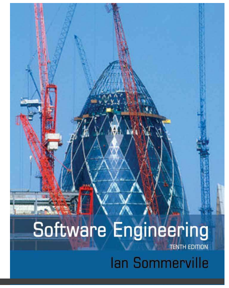


- ✧**Scattering**
  - The specification of one property is **not encapsulated** in a single requirements unit, e.g., a viewpoint, a use case.

- **Tangling**
  - Each requirements unit contains descriptions of **several properties** or different functionalities

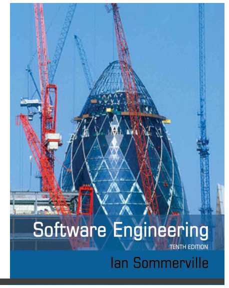


Potential Benefits of AOSD



- ✧ Improved ability to reason about problem domain and corresponding solution
- ✧ Reduction in application code size, development costs and maintenance time
- ✧ Improved code reuse
- ✧ Requirements, architecture and design-level reuse
- ✧ Improved ability to engineer product lines
- ✧ Context-sensitive application adaptation
- ✧ Improved modelling methods

Crosscutting: The Tracing Concern




```
class A {
  // some attributes
  void m1( ) {
     System.out.println("Entering A.m1( )");
     // method code
     System.out.println("Leaving A.m1( )");
  }

  String m2( ) {
     System.out.println("Entering A.m2( )");
     // method code
     System.out.println("Leaving A.m2( )");
     // return a string
  }
```

```
class B {
  // some attributes
  void m2( ) {
     System.out.println("Entering B.m2( )");
     // method code
     System.out.println("Leaving B.m2( )");
  }

  int m3( ) {
     System.out.println("Entering B.m3( )");
     // method code
     System.out.println("Leaving B.m3( )");
     // return an integer
  }
```

Wouldn't it be Nice if …

```
class A {
   // some attributes
   void m1( ) {
      // method code
}

   String m2( ) {
      // method code
      // return a string
   }
```

```
class B {
   // some attributes
   void m2( ) {
      // method code
      }

   int m3( ) {
      // method code
      // return an integer
   }
```

aspect Tracing {

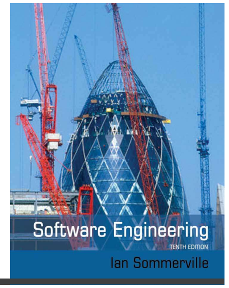
when someone ***calls*** these methods

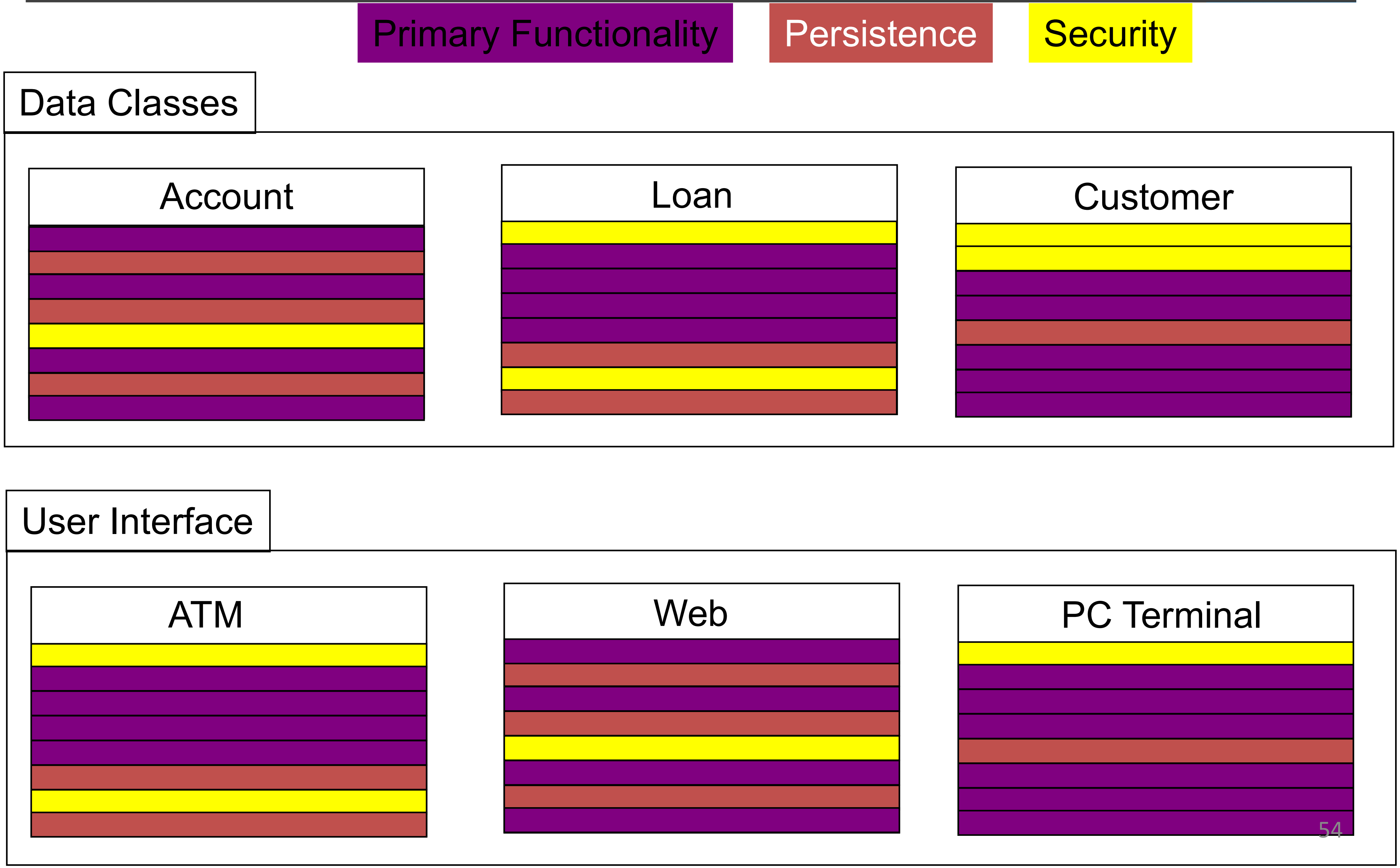
before the call {System.out.println("Entering " + methodSignature);}

after the call {System.out.println("Leaving " + methodSignature);}

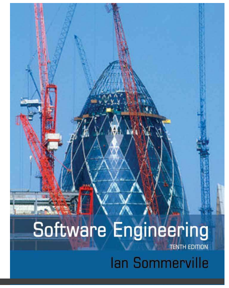
}

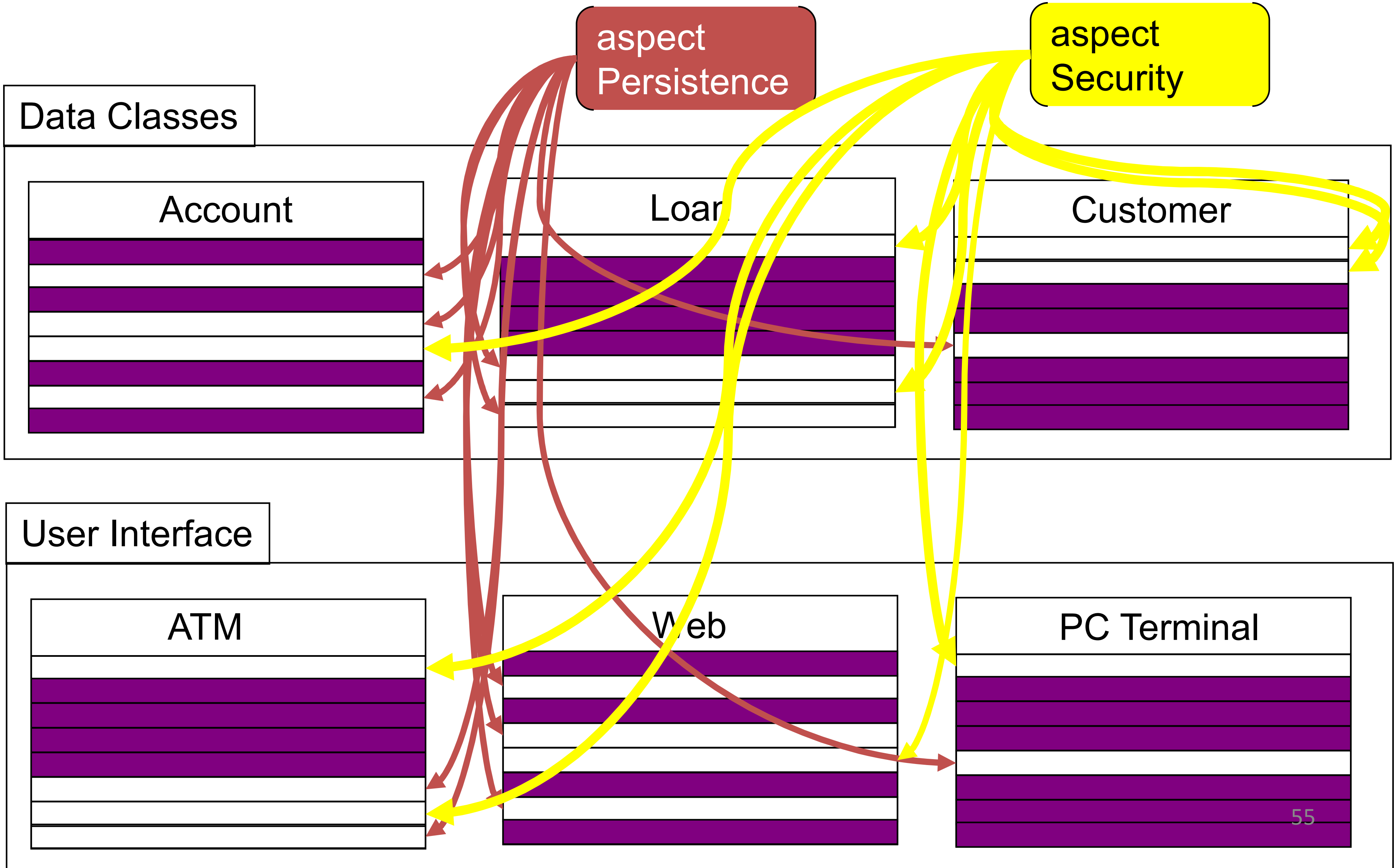
Tangling and Scattering: the Bank Example

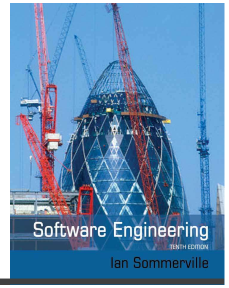




Wouldn't it be Nice if …







Main Value of Aspect-Orientation

- **Abstraction**: abstract away from the details of how that crosscutting concern, or *aspect*, might be scattered and tangled with the functionality of other modules in the system
- **Modularization**: keep crosscutting concerns separated regardless of how they affect or influence various other modules in the system, so then we can reason about each module in isolation – **Modular Reasoning**
- **Composition**: the various modules need to relate to each other in a systematic and coherent fashion so that one may reason about the global or emergent properties of the system – **Compositional Reasoning**

Application wrapping