

Semantic Web

Juha Puustjärvi

Course books:

- M.C. Daconta, L.J. Obrst, and K.T. Smith. The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management. Wiley Publishing, 2003.
- G. Antoniou, and F. Harmelen. A semantic Web Primer. The MIT Press, 2004.
- M. P. Singh, and M. H. Huhns. Service-Oriented Computing: Semantics, Processes, Agents. John Wiley & Sons. 2005.

Contents

- Chapter 1: Today's Web and the Semantic Web
- Chapter 2: The Business Case for the Semantic Web
- Chapter 3: Understanding XML and its Impact on the Enterprise
- Chapter 4: Understanding Web Services
- Chapter 5: Understanding Resource Description Framework
- Chapter 6: Understanding XML Related Technologies
- Chapter 7: Understanding Taxonomies
- Chapter 8: Understanding Ontologies
- Chapter 9: Semantic Web Services
- Chapter 10: An Organization's Roadmap to Semantic Web

Chapter 1: Today's Web and the Semantic Web

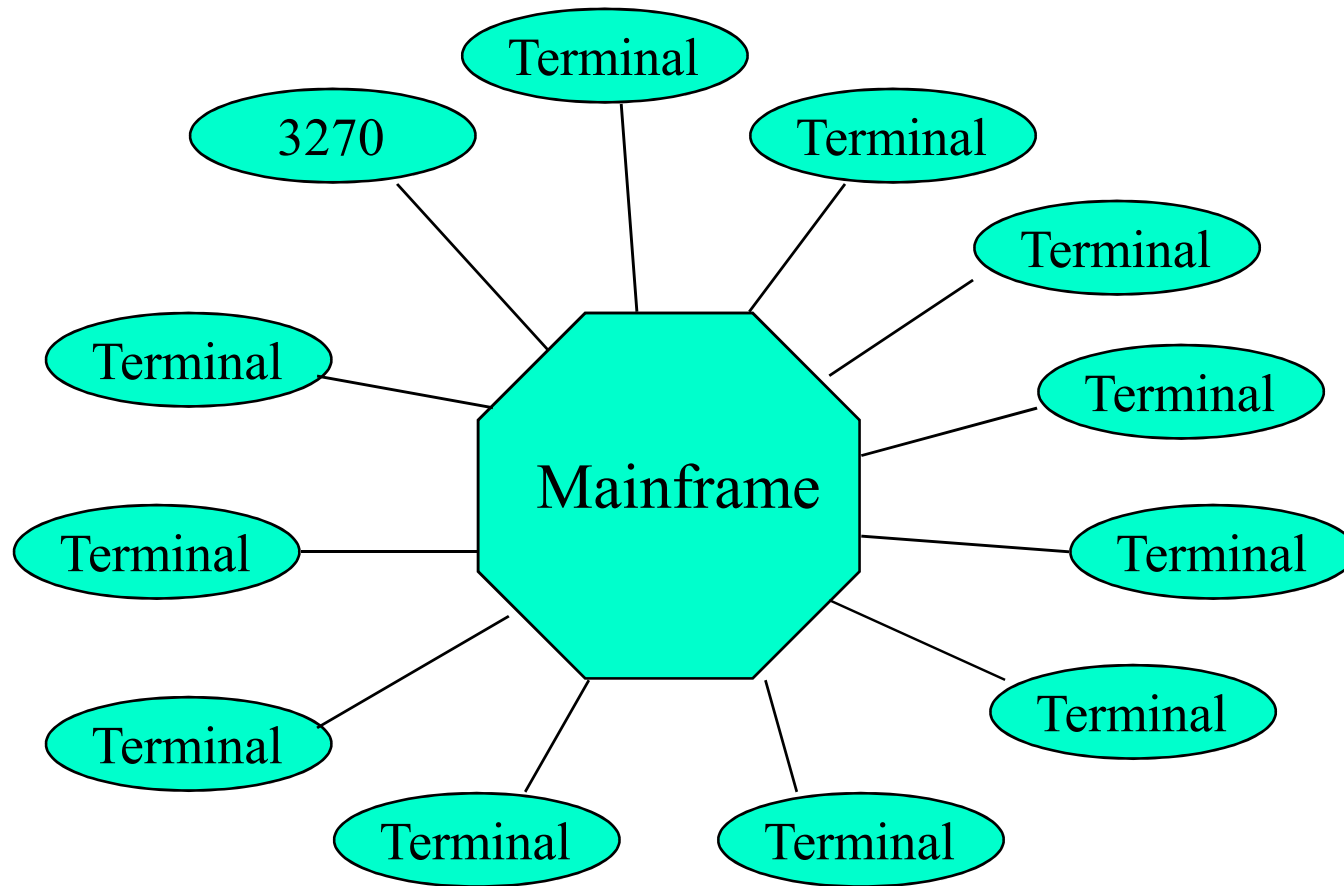
Brief History of Information Technology

- First-generation information systems provide centralized processing that is controlled and accessed from simple terminals that have no data-processing capabilities of their own.
- Second-generation information systems are organized into servers that provide general-purpose processing, data, files, and applications and clients that interact with the servers and provide special-purpose processing, inputs, and outputs.

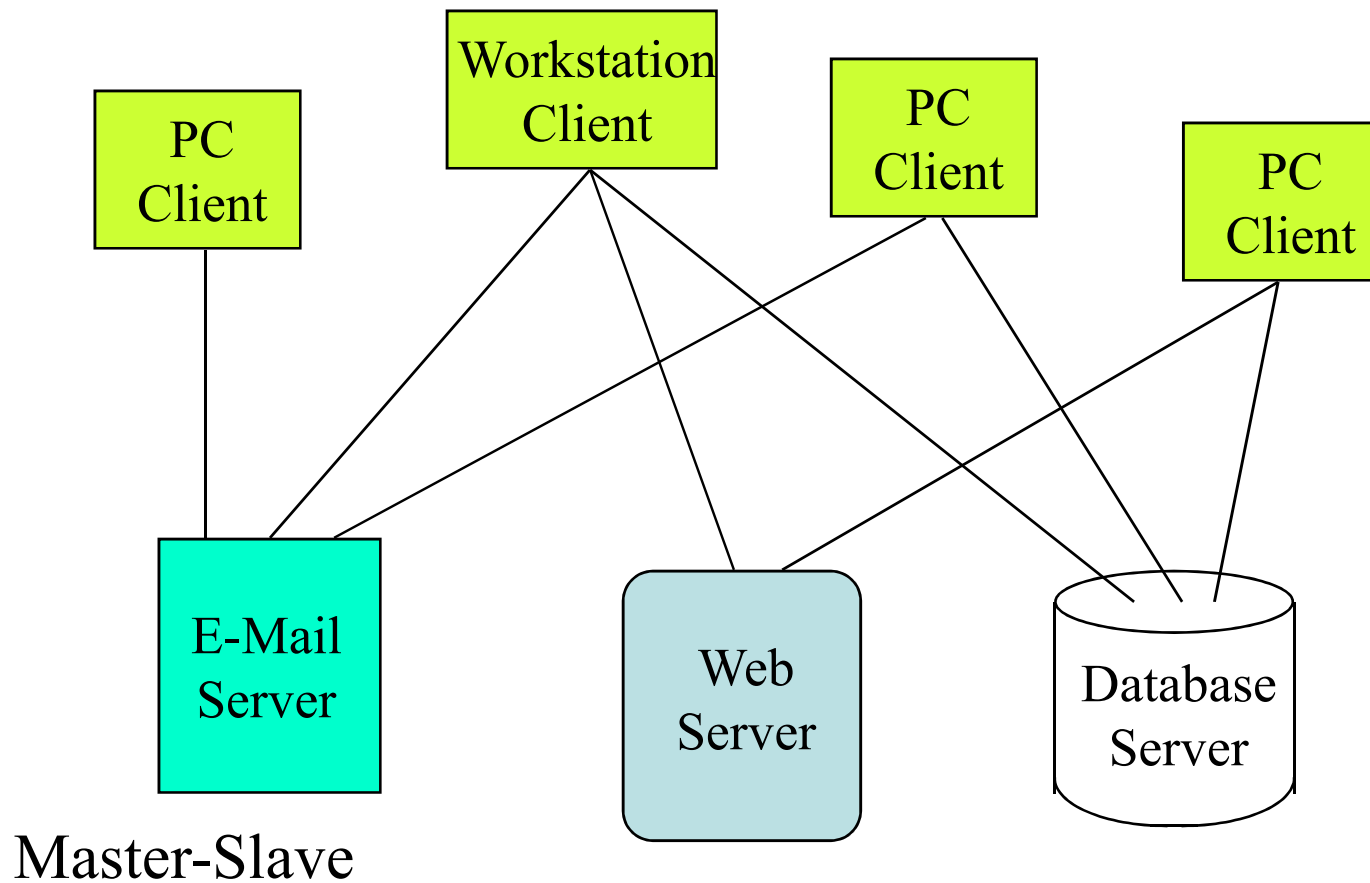
Brief History of Information Technology

- Third-generation information systems, which include those termed peer-to-peer, enable each node of a distributed set of processors to behave as both a client and a server; they may still use some servers.
- Emerging next-generation information systems are cooperative, where autonomous, active, heterogeneous components enable the components collectively to provide solutions.

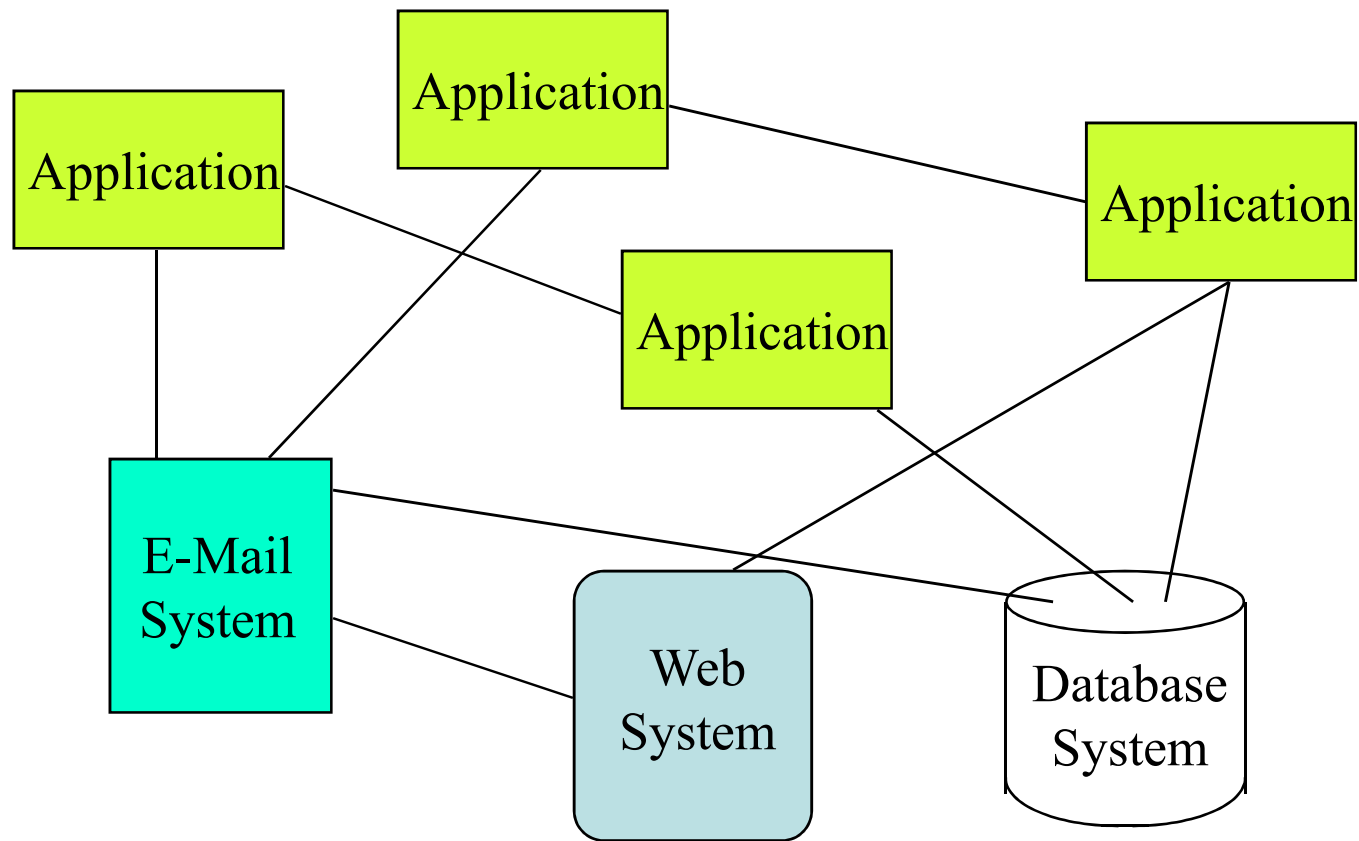
System Architectures: Centralized



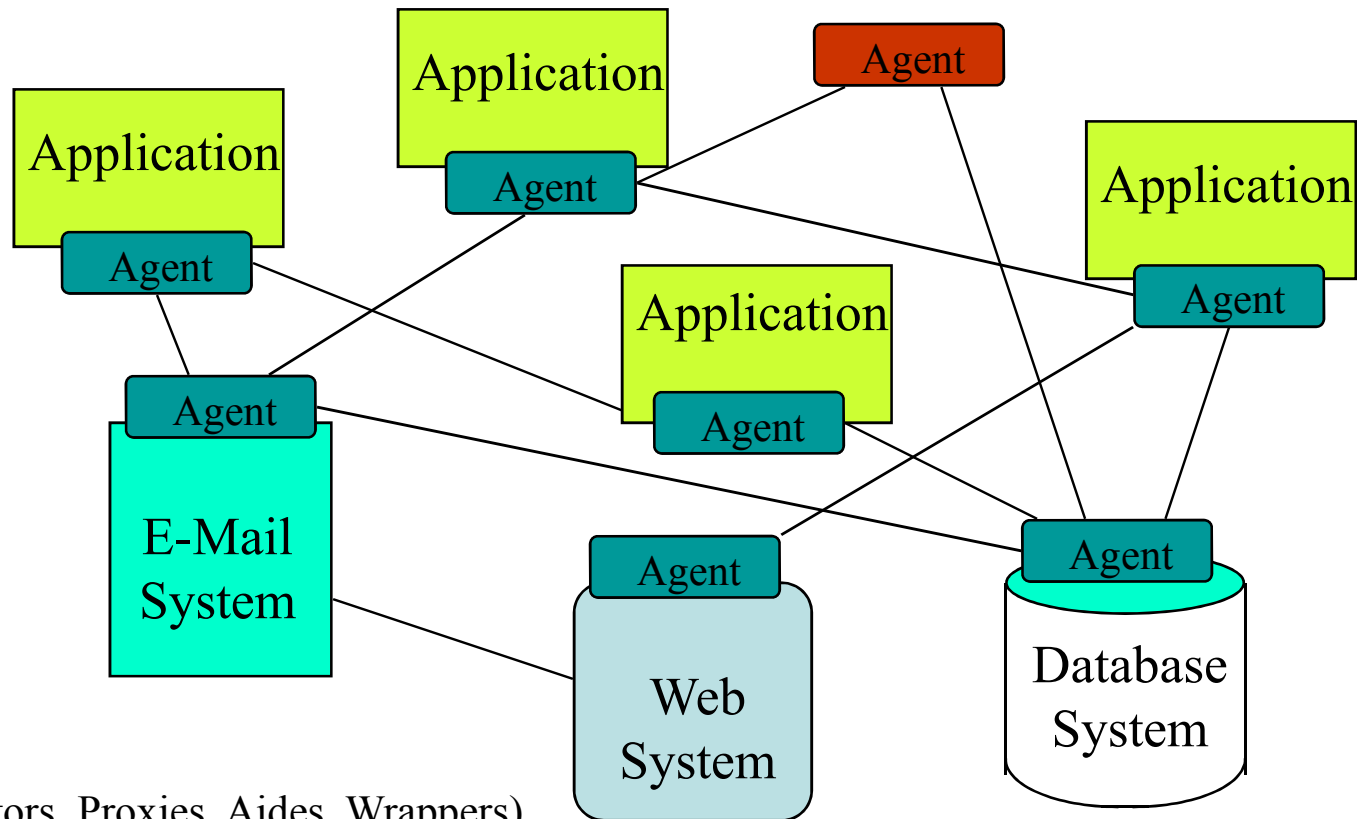
System Architectures: Client-Server



System Architectures: Peer-to-Peer



System Architectures: Cooperative



(Mediators, Proxies, Aides, Wrappers)

Open Environments

- The term *open* implies that the components involved are *autonomous* and *heterogeneous*, and system configurations can change *dynamically*.
- Often, we would want to constrain the design and behavior of components, and thus limit the openness of the system – however, the system would still have to deal with the rest of the world, which would remain open.
 - E.g., a company might develop an enterprise integration system that is wholly within the enterprise – yet the system would have to deal with external parties, e.g., to handle supply and production chains.

Autonomy

- Autonomy means that the components in an environment function solely under their own control.
 - E.g., an e-commerce site may or may not remove some item from its catalog.
- Software components are autonomous because they reflect the autonomy of the human and corporate interests that they represent on the Web,
 - i.e., there are sociopolitical reasons for autonomy
- A consequence of autonomy is that updates can occur only under local control.

Heterogeneity

- Heterogeneity means that the various components of given system are different in their design and construction.
 - Often the reasons are historical: components fielded today may have arisen out of legacy systems that were initially constructed for different narrow uses, but eventually expanded in their scopes to participate in the same system.
- Heterogeneity can arise at a variety of levels in a system, such as networking protocols, encodings of information, and data formats.
 - Clearly, standardization at each level reduces heterogeneity and can improve productivity through enhanced interoperability.
 - This is the reason that standards such as the Internet Protocol (IP), HTTP, UCS Transportation Format (UTF-8), and XML have gained currency.

- Heterogeneity also arises at the level of semantics and usage, where it may be hardest to resolve and sometimes even to detect.
- However, there is a reason why heterogeneity emerges and should be allowed to persist.
 - To remove heterogeneity would involve redesigning and re-implementing the various components.

Dynamism

- An open environment can exhibit dynamism in two main respects.
 - First, because of autonomy, its participants can behave arbitrary.
 - Second, they may also join or leave an open environment on a whim.

The Challenges of Open Environments

- Open environments pose significant technical challenges.
- In particular, the developed approaches must
 - cope with the scale of the number of participants,
 - respect the autonomy, and
 - accommodate the heterogeneity of the various participants,while maintaining coordination

Services

- Just like objects a generation ago, services is now the key buzzword. However, services mean different things to different people:
 - A piece of business logic accessible via the Internet using open standards (Microsoft).
 - Encapsulated, loosely coupled, contracted software functions, offered via standard protocols over the Web (DestiCorp).
 - Loosely coupled software components that interact with one another dynamically via standard Internet technologies (Gartner).
 - A software application identified by a URL, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts and supports direct interaction with other software applications using XML-based messages via Internet-based protocols (W3C).

A historical view of services over the Web

Generation	Scope	Technology	Example
First	All	Browser	Any HTML page
Second	Programmatic	Screen scraper	Systematically generated HTML content
Third	Standardized	Web services	Formally described service
Fourth	Semantic	Semantic Web services	Semantically described service

Related Standards Bodies

- Since services involve serious work and interactions among the implementations and systems of diverse entities, it is only natural that several technologies related to services would be standardized.
 - As in much of computer science, standardization in services often proceeds in a *de facto* manner, where a standard is established merely by fact of being adopted by a large number of vendors and users.
 - Standards bodies take the lead in coming up with *de jure* standards, and clean up and formalize emerging *de facto* standards.

- The following are the most important standards bodies and initiatives for services.
 - **IETF.** The Internet Engineering Task Force is charged with the creation and dissemination of standards dealing with Internet technologies. Besides the TCP/IP suite and URI's it is responsible for HTTP, Session Initiation Protocol SIP and SMTP.
 - **OMG.** The Object Management Group has been developing standards for modeling, interoperating, and enacting distributed object systems. Its most popular standards include UML and CORBA.
 - **W3C.** The World-Wide Web Consortium is an organization that promotes standards dealing with Web technologies. The W3C has mostly emphasized the representational aspects of the Web, deferring to other bodies for networking and other computational standards, e.g., those involving transactions. W3C's main standards of interest for services include XML, XML Schema, WSDL, SOAP, and WSCI.

- **OASIS.** The Organization for the Advancement of Structured Information Standards standardizes a number of protocols and methodologies relevant to Web services including the Universal Business Language UBL, UDDI and Business Process Specification Language for Web Services (BPEL4WS), and in collaboration with UN/CEFACT, ebXML.
- **UN/CEFACT.** The Nations Center for Trade Facilitation and Electronic Business focuses on the facilitation of international transactions, through the simplification and harmonization of procedures and information flow. ebXML is one of its development.
- **WS-I.** The Web Services Interoperability Organization is an open, industry organization chartered to promote the interoperability of Web services across platforms, operating systems, and programming languages. Its primary contribution is Basic Profile version 1.0 (BP 1.0).

- **BPMI.org.** The Business Process Management Initiative is working to standardize the management of business processes that span multiple applications, corporate departments, and business partners. It integrated XLANG (Microsoft) and WSFL (IBM) into BPEL4WS.
- **WFMC.** The Workflow Management Coalition develops standardized models for workflows, and workflow engines, as well as protocols for monitoring and controlling workflows.
- **FIPA.** The foundation for Intelligent Physical Agents promotes technologies and specifications that facilitate the end-to-end interoperation of intelligent agent systems for industrial applications. Its standards include agent management technologies and agent communication languages.

Standards for Web Services

- There have been several major efforts to standardize services and service protocols, particularly for electronic business.
- The relationship of the different proposed standards and methodologies for automating electronic business is presented in the following figure.

UDDI				ebXML Registries	Discovery
				ebXML CPA	Contracts and agreements
OWL-S Service Model		BPEL4WS		BPML	Process and workflow orchestrations
		WS-AtomicTransaction and WS-BusinessActivity		BTP	QoS: Transactions
OWL-S Service Profile		WS-Reliable Messaging	WS-Coordination	WSCI	QoS: Choreography
OWL-S Service Grounding		WS-Security	WSCL		QoS: Conversations
OWL	PSL	WS-Policy	WSDL		QoS: Service descriptions and bindings
RDF	SOAP			ebXML messaging	Messaging
XML, DTD, and XML Schema					Encoding
HTTP, FTP, SMTP, SIP, etc.					Transport

- The efforts to standardize services and service protocols, particularly for electronic business, have resulted the rightmost stack.
- The leftmost stack is the result of development efforts by the Semantic Web research community in conjunction with the W3C.
- The central stack is primarily the result of standards efforts led by IBM, Microsoft, BEA, HP, and Sun Microsystems.
 - In general, these have been separate from standards bodies, but will be ratified eventually by one or more appropriate such bodies.

- Each stack makes use of the following abstraction levels:
 - The ***transport layer*** provides the fundamental protocols for communicating information among the components in a distributed system of services.
 - The ***encoding layer*** (XML-layer) is the foundation for interoperation among enterprises and for the envisioned Semantic Web. The standards of this level describes the grammars for syntactically well formed data and documents.
 - The ***messaging layer*** describes the formats using which documents and services invocations are communicated.
 - The ***service description and bindings layer*** describes the functionality of Web services in terms of their implementations, interfaces, and results.

- A **conversation** is an instance of a protocol of interactions among services, describing the sequence of documents and invocations exchanged by an individual service.
- **Choreography** protocols coordinate collections of Web services into patterns that provide a desired outcome.
- **Transaction protocols** specify not only the behavioral commitments of the autonomous components, but also the means to rectify the problems that arise when exceptions and commitment failures occur.
- The **orchestration layer** has protocols for workflows and business processes, which are composed of more primitive services and components. Orchestration implies a centralized control mechanism, whereas choreography does not.

- ***Contracts*** and ***agreements*** formalize commitments among autonomous components in order to automate electronic business and provide outcomes that have legal force and consequences.
- The ***discovery layer*** specifies the protocols and languages needed for services to advertise their capabilities and for clients that need such capabilities to locate and use the services.

Today's Web and the Semantic Web

- Today's Web
 - WWW has changed the way people communicate with each others and the way business is conducted
 - WWW is currently transforming the world toward a knowledge society
 - Computers are focusing to the entry points to the information highways
 - Most of today's Web content is suitable for human consumption
 - Keyword-based search engines (e.g., Google) are the main tools for using today's Web

The problems of the keyword-based search engines

- High recall, low precision
- Low or no recall

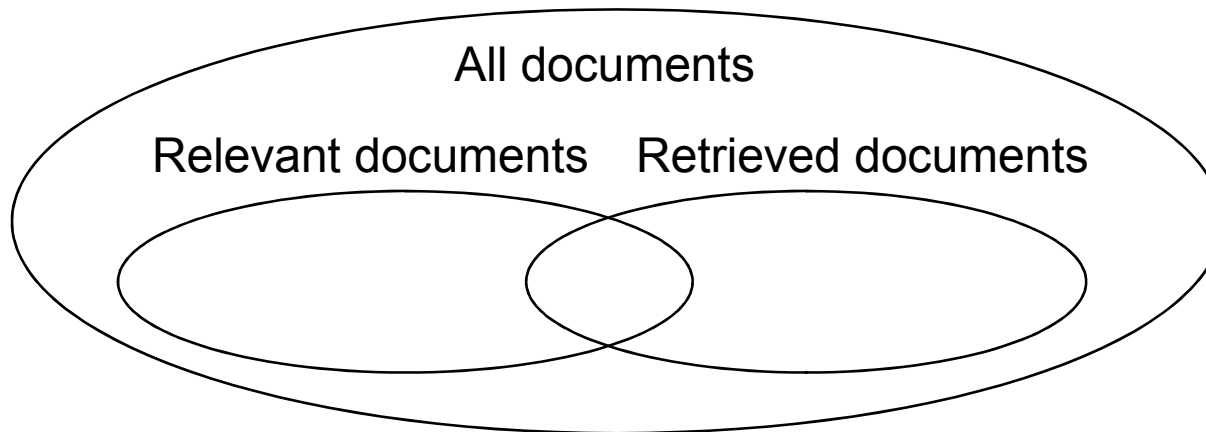


Figure. Relevant documents and retrieved documents.

The problems of the keyword-based search engines

- Results are highly sensitive to vocabulary
 - Often initial keywords do not get the results we want; in these cases the relevant documents use different terminology from the original query
- Results are single web pages
 - If we need information that is spread over various documents, we must initiate several queries to collect the relevant documents, and then we must manually extract the partial information and put it together

Note: The term Information retrieval used with search engine is somehow misleading; *location finder* is more appropriate term. Search engines are also typically isolated applications, i.e., they are not accessible by other software tools.

The problems of the keyword-based search engines, continues...

- The meaning of Web content is not *machine – accessible, e.g.,*

It is difficult to distinguish meaning of

I am a professor of computer science

from

I am a professor of computer science, you may think.

From Today's Web to the Semantic Web: Examples

- Knowledge management
 - Knowledge management concerns itself with acquiring, accessing and maintaining knowledge within an organization
 - Has emerged as a key activity of large business because they view internal knowledge as an *intellectual asset* from which they can draw greater productivity, create new value, and increase their competitiveness
 - Knowledge management is particularly important for international organizations with geographically dispersed departments

- From knowledge management point of view the current technology suffers from limitations in the following areas:
 - Searching information
 - Companies usually dependent on search engines
 - Extracting information
 - Human time and effort are required to browse the retrieved documents for relevant information
 - Maintaining information
 - Currently there are problems, such as inconsistencies in terminology and failure to remove outdated information
 - Uncovering information
 - New knowledge implicitly existing in corporate database is extracted using data mining
 - Viewing information
 - Often it is desirable to restrict access to certain information to certain groups of employees. “Views” are hard to realize over Intranet or the Web

- The aim of the Semantic Web is to allow much more advanced knowledge management system:
 - Knowledge will be organized in conceptual spaces according to its meaning
 - Automated tools will support maintenance by checking for inconsistencies and extracting new knowledge
 - Keyword based search will be replaced by query answering: requested knowledge will be retrieved, extracted, and presented in a human-friendly way
 - Query answering over several documents will be supported
 - Defining who may view certain parts of information (even parts of documents) will be possible.

Business-to-Consumer Electronic Commerce (B2C)

- B2C electronic commerce is the predominant commercial experience of Web users
 - A typical scenario involves a user's visiting one or several shops, browsing their offers and ordering products
 - Ideally, a user would collect information about prices, terms, and conditions (such as availability) of all, or at least all major, online shops and then proceed to select the best offer. However, manual browsing is too time-consuming.
 - To alleviate this situation, tools for shopping around on the Web are available in the form of shopbots, software agents that visit several shops extract product and price information, and compile a market overview.
 - The function of shopbots are provided by *wrappers*, programs that extract information from an online store. One wrapper per store must be developed.
 - The information is extracted from the online store site through keyword search and other means of textual analysis

Business-to-Consumer Electronic Commerce (B2C)

- The Semantic Web will allow the development of software agents that can *interpret* the product information and the terms of service
 - Pricing and product information will be extracted correctly, and delivery and privacy policies will be interpreted and compared to the user requirements
 - Additional information about the reputation of online shops will be retrieved from other sources, for example. Independent rating agencies or consumer bodies
 - The low-level programming of wrappers will become obsolete
 - More sophisticated shopping agents will be able to conduct automated negotiations, on the buyer's behalf, with shop agents

Business-to-Business Electronic Commerce (B2B)

- The greatest economic promise of all online technologies lies in the area of B2B
- Traditionally business have exchanged their data using the Electronic Data Interchange (EDI) approach
 - EDI-technology is complicated and understood only by experts
 - Each B2B communication requires separate programming
 - EDI is also an isolated technology in the sense that interchanged data cannot be easily integrated with other business applications
- Business have increasingly been looking at Internet-based solutions, and new business models such as B2B-portals have emerged, still B2B commerce is hampered by the lack of standards

Business-to-Business Electronic Commerce (B2B)

- The new standard of XML is a big improvement but can still support communications only in cases where there is a priori agreement on the vocabulary to be used and on its meaning
- The realization of The Semantic Web will allow businesses to enter partnerships without much overhead
- Differences in terminology will be resolved using standard abstract domain models, and data will be interchanged using translation services
- *Auctioning, negotiations, and drafting contracts* will be carried out automatically or semi-automatically by software agents

Explicit metadata

- Currently, Web content is formatted for human readers rather than programs.
- HTML is the predominant language in which Web pages are written directly or using tools
- A portion of a typical HTML-based Web page of a physical therapist might look like the following

“HTML” example

<h1>Agilitas Physiotherapy Centre</h1>

Welcome to the home page of the Agilitas Physiotherapy Centre....

<h2>Consultation hours</h2>

Mon 11 am -7 pm

Tue 11am – 7 pm

Wed 3 am – 7pm

Thu 10 am – 8 pm

Fri 11am – 4 pm <p>

But note that we do not offer consultation during the weeks of the
State of origingames.

Note. For people the information is presented in a satisfactory way, but machines will have their problems, e.g., finding the exact consultation hours, i.e., when there are no games.

“XML” example

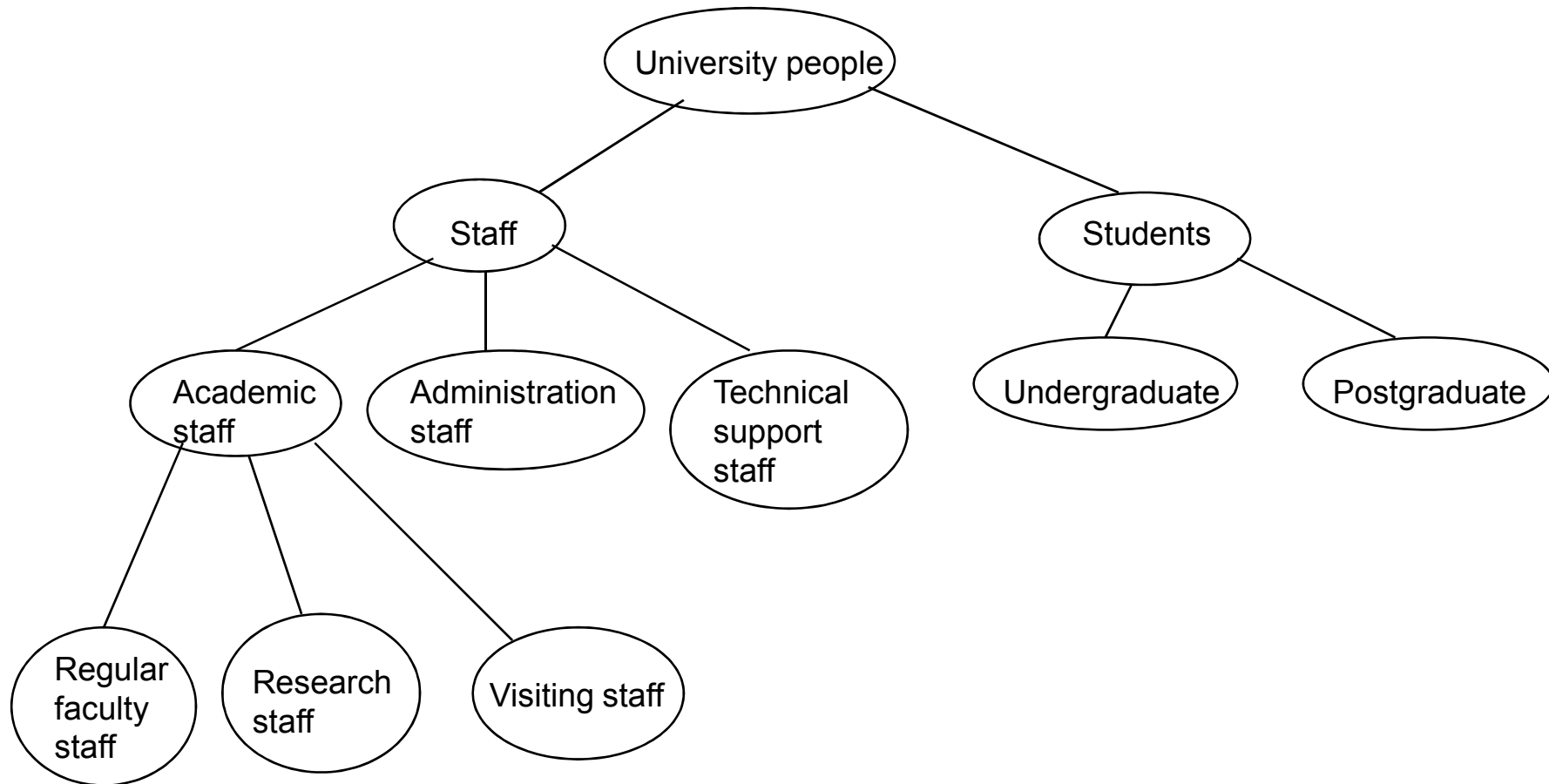
```
<company>  
  <treatmentOffered>Physiotherapy</treatmentOffered>  
  <companyName>Agilitas Physiotherapy Centre</companyName>  
  <staff>  
    <therapist>Lisa Davenport</therapist>  
    <therapist>Steve Matthews</therapist>  
    <secretary>Kelly Townsend</secretary>  
  </staff>  
</company>
```

Note: This representation is far more processable by machines.

Ontologies

- The term *Ontology* originates from philosophy “the study of the nature of existence”
- For our purpose we use the definition “An ontology is an explicit and formal specification of a conceptualization”
- In general, an ontology describes formally a domain of discourse
 - Typically an ontology consists of a finite list of terms and the relationship between these terms
 - The terms denote important *concepts* (classes or objects) of the domain, e.g., in the university setting staff members, students, course and disciplines are some important concepts
 - The *relationships* typically include hierarchies of classes
 - A hierarchy specifies a class *C* to be a subclass of an other class *C'* if every object in *C* is also included in *C'*

An example hierarchy



Apart from subclass relationships, ontologies may include information such as:

- properties,
 - e.g., X teaches Y
- value restrictions,
 - e.g., only faculty members can teach courses
- disjointness statements,
 - e.g., faculty and general staff are disjoint
- specification of logical relationships between objects,
 - e.g., every department must include at least ten faculty members

- In the context of Web, ontologies provide a *shared understanding of a domain*
- A shared understanding is necessary to overcome differences in terminology
 - One application's zip code may be the same as another application's area code
 - Two applications may use the same term with different meanings, e.g., in university A, a course may refer to a degree (like computer science), while in university B it may mean a single subject , e.g. CS 100
- Differences can be overcome by mapping the particular terminology to a shared ontology or by defining direct mapping between the ontologies;
 - in either case ontologies support semantic interoperability

Ontologies are also useful for:

- the organization and navigation of Web sites
 - Many Web sites expose on the left-hand side of the page the top levels of concept hierarchy of terms. The user may click on one of them to expand the subcategories
- improving the accuracy of Web searches
 - The search engine can look for pages that refer to a precise *concept* in an ontology instead of collecting all pages in which certain, generally ambiguous, keywords occur. In this way differences in terminology between Web pages and the queries can be overcome
- exploiting generalization /specialization information in Web searches
 - If a query fails to find any relevant documents, the search engine may suggest to the user a more general query. Also if too many answers are retrieved, the search engine may suggest to the user some specification

- In Artificial intelligence (AI) there is a long tradition of developing ontology languages
 - It is a foundation Semantic Web research can build on
- At present, the most important ontology languages for the Web are the following
 - **XML** provides a surface syntax for structured documents but impose no semantic constraints on the meaning of these documents
 - **XML Schema** is a language for restricting the structure of XML documents

- **RDF** is a data model for objects (“resources”)and relations between them; it provides a simple semantics for this data model; and these data models can be represented in an XML syntax
- **RDF Schema** is a vocabulary description language for describing properties and classes of RDF resources, with a semantics for generalization hierarchies of such properties and classes
- **OWL** is richer vocabulary language for describing properties and classes, such as relations between classes (e.g., disjointness), cardinality (e.g., exactly one), equality, richer typing properties, characteristics of properties (e.g., symmetry), and enumerated classes

Logic

- Logic is the discipline that studies the principle of reasoning; it goes back to Aristotle
 - logic offers formal languages for expressing knowledge
 - logic provides us with *well-understood formal semantics*
 - In most logics, the meaning of sentences is defined without the need to operationalize the knowledge
 - Often we speak of declarative knowledge: we describe *what* holds without caring about *how* it can be deduced
 - automated reasoners can deduce (infer) conclusions from the given knowledge, thus making implicit knowledge explicit (such reasoners have been studied extensively in AI)

Example of inference in logic

Suppose we know that all professors are faculty members, that all faculty members are staff members, and that Michael is a professor

In predicate logic this information is expressed as follows:

$\text{prof}(X) \rightarrow \text{faculty}(X)$

$\text{faculty}(X) \rightarrow \text{staff}(X)$

$\text{prof}(\text{Michael})$

Then we can deduce the following

$\text{faculty}(\text{Michael})$

$\text{staff}(\text{Michael})$

$\text{prof}(X) \rightarrow \text{staff}(X)$

Note. This example involves knowledge typically found in ontologies. Thus logic can be used to uncover knowledge that is implicitly given.

Example of inference in logic

- Logic is more general than ontologies; it can also be used by intelligent agents for making decisions and selecting courses of action.
- For example a shop agent may decide to grant a discount to a customer based on the rule

$\text{loyal}(\text{Customer}(X)) \rightarrow \text{discount}(5\%)$

Where the loyalty of customers is determined from data stored in the corporate database

Note. Generally there is trade-of between expressive power and computational efficiency: the more expressive a logic is, the more computationally expensive it becomes to draw conclusions. And drawing certain conclusions may become impossible if noncomputability barriers are encountered.

- Most knowledge relevant to the Semantic Web seems to be of a relatively restricted form,
 - e.g., the previous examples involved *rules* of the form

if condition then conclusion

and only finitely many objects needed to be considered. This subset of logic is tractable and is supported by efficient reasoning tools.

Propositional logic

- Propositional logic is the simplest kind of logic
- Enables formally express simple semantic truths about the world called propositions
- A proposition is an expression (statement) in logic about the world or some part of it that is either true or false (in certain logics also unknown)
- A limitation of propositional logic is that one cannot speak about individuals (instances like John, who is an instance of a management employee) because the granularity is not fine enough
 - The basic unit is the proposition, which is either true or false
 - One cannot “get inside” the proposition and pull out instances or classes or properties (for these one needs first-order predicate logic)

Propositional logic example

PROPOSITIONS IN ENGLISH	PROPOSITIONS IN PROPOSITIONAL LOGIC
<p>If John is a management employee, then John manages an organization</p>	$p \rightarrow q$
<p>John is a management employee</p> <hr style="width: 20%; margin-left: 0;"/>	<p>p</p>
<p>John manages an organization</p>	<hr style="width: 20%; margin-left: 0;"/> <p>q Modus ponens</p>
<p>Modus ponens</p>	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <p style="font-size: 2em;">}</p> <p>Assertions</p> </div> <div style="margin-right: 10px;"> <p style="font-size: 2em;">}</p> <p>Conclusion</p> </div> <div> <p style="font-size: 3em;">}</p> <p>Proof</p> </div> </div>

The way to read a proof: if the assertions are held to be true, it follows logically from them that the conclusion is true – and true by reason of a logical inference rule, here the rule *modus ponens*

First –order predicate logic

- A ***predicate*** is a feature of language (and logic) that can be used to make a statement or attribute a property to something, e.g., properties of being ***a management employee*** and ***managing an organization***
- An instantiated predicate is a proposition, e.g., *managrment_employee(john) = true*
- An uninstantiated predicate, e.g., *management_employee(x)* is not a proposition because the statement does not have a truth value

Predicate logic example

PROPOSITIONS AND PREDICATES IN ENGLISH

If John is a management employee,
then John manages an organization

John is a management employee

John manages an organization

Modus ponens

PROPOSITIONS AND PREDICATES IN FIRST-ORDER PREDICATE LOGIC

$p(x) \rightarrow q(x)$

$P(\text{John})$

$q(\text{John})$ **Modus ponens**

Using quantifiers in predicate logic

- A quantifier is a logical symbol that enables one to quantify over instances or individuals
 - Universal quantifier means *All*
 - Existential quantifier means *Some*
- *Ordinary predicate logic* is called *first-order* as it only quantifies over instances
- *Second order logics* quantify over both instances and predicates

Example of quantifiers in Predicate Logic

PROPOSITIONS AND PREDICATES IN ENGLISH

**Everyone who is a management
employee manages some organization**

Or:

**For everyone who is a management
employee, there is some organization
that that person manages**

PROPOSITIONS AND PREDICATES IN FIRST-ORDER PREDICATE LOGIC

$\text{All } x.[p(x) \rightarrow \text{some } y.[q(y) \wedge r(x,y)]]$

”for all x, if x is a p,

then there is some y such that

y is a q and x is in the r relation to y”

Logical theory behind DAML+OIL and OWL

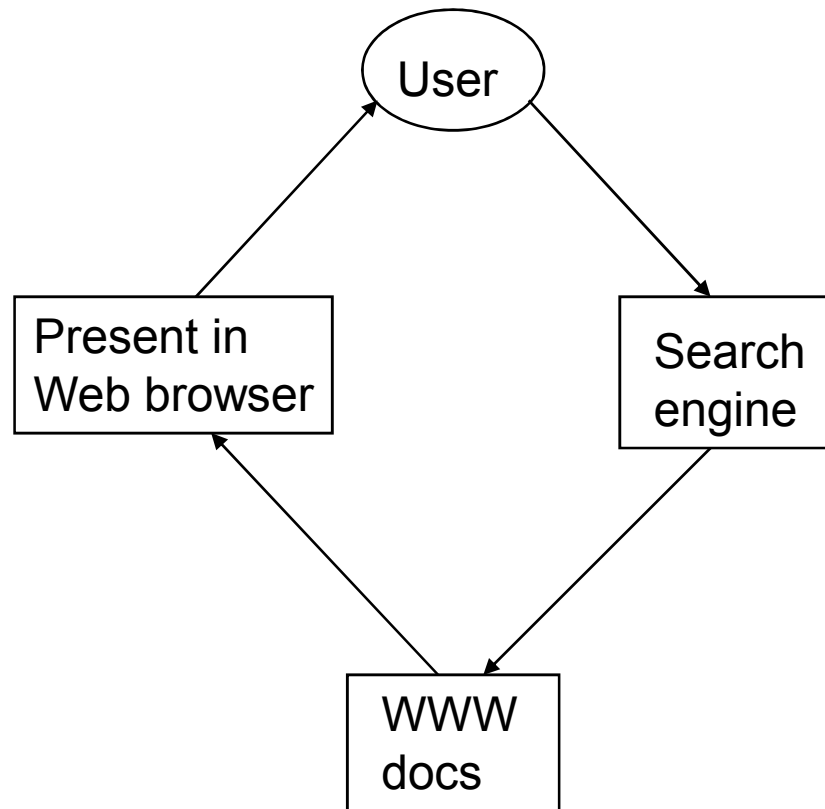
- Ontologies that use logical theories are modeled in semantic languages such as DAML+OIL and OWL
- The logic behind DAML+OIL and OWL is almost but not quite as complicated as first-order predicate logic (description logics explicitly try to achieve a good trade-off between semantic richness and machine tractability)
- The use of ontology development tools based on DAML+OIL or OWL does not require the understanding of formal logics

Agents in the Semantic Web

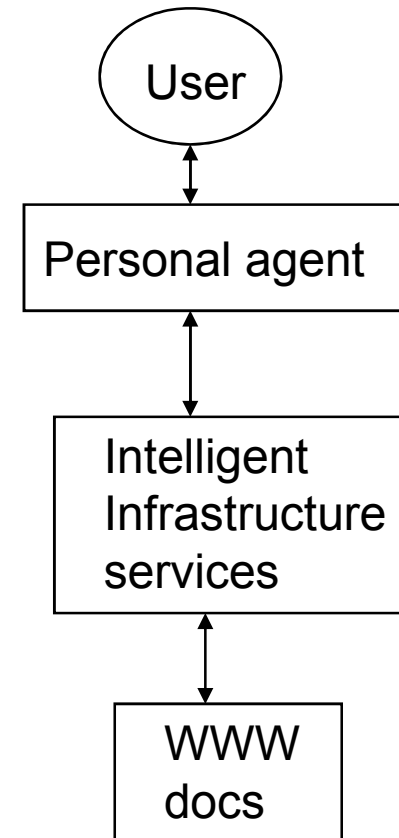
- Agents are pieces of software that work autonomously and proactively
- Conceptually they evolved out of the concepts of object-oriented programming and component-based software development
- A personal agent on the Semantic Web will receive some tasks and preferences from the person,
 - seek information from Web sources,
 - communicate with other agents,
 - compare information about user requirements and preferences,
 - select certain choices, and
 - give answers to the user

Intelligent personal agents

Today



In the future



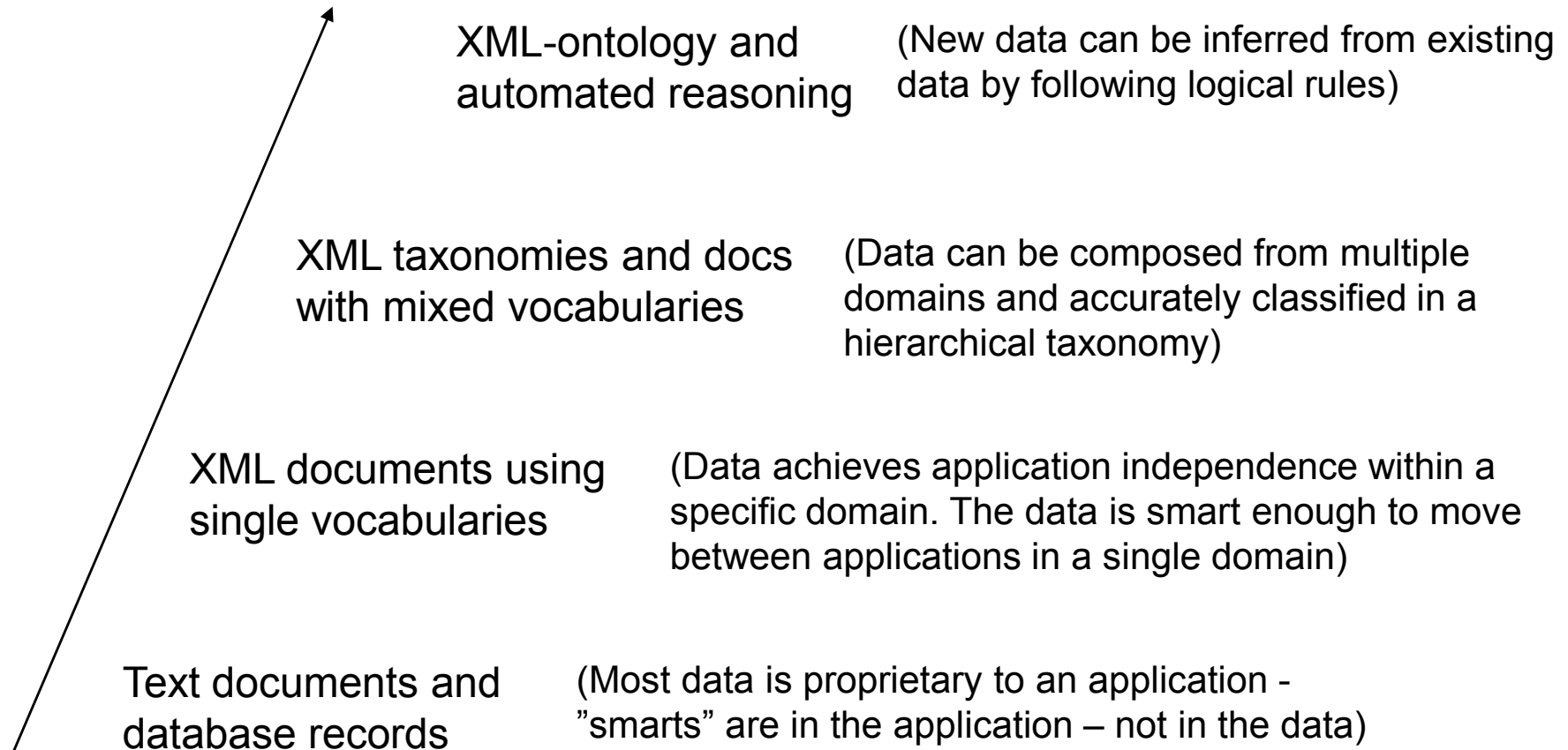
- Agents will not replace human users on the Semantic Web, nor will they necessarily make decisions
 - The role of agents will be to collect and organize information, and present choices for the users to select from
- Semantic web agents will make use of many technologies including:
 - Metadata will be used to identify and extract information from Web sources
 - Ontologies will be used to assist in Web searches, to interpret retrieved information, and to communicate with other agents
 - Logic will be used for processing retrieved information and for drawing conclusions

What is a Semantic Web

- Tim Berners-Lee has a two-part vision for the future of the Web
 - The first part is to make the Web a more collaborative medium
 - The second part is to make the Web understandable, and thus processable, by machines
- A definition of the Semantic Web:
a machine processable web of smart data
 - Smart data:
data that is application-independent, composeable, classified, and part of a larger information ecosystem

The path to machine-processable data is to make the data smarter

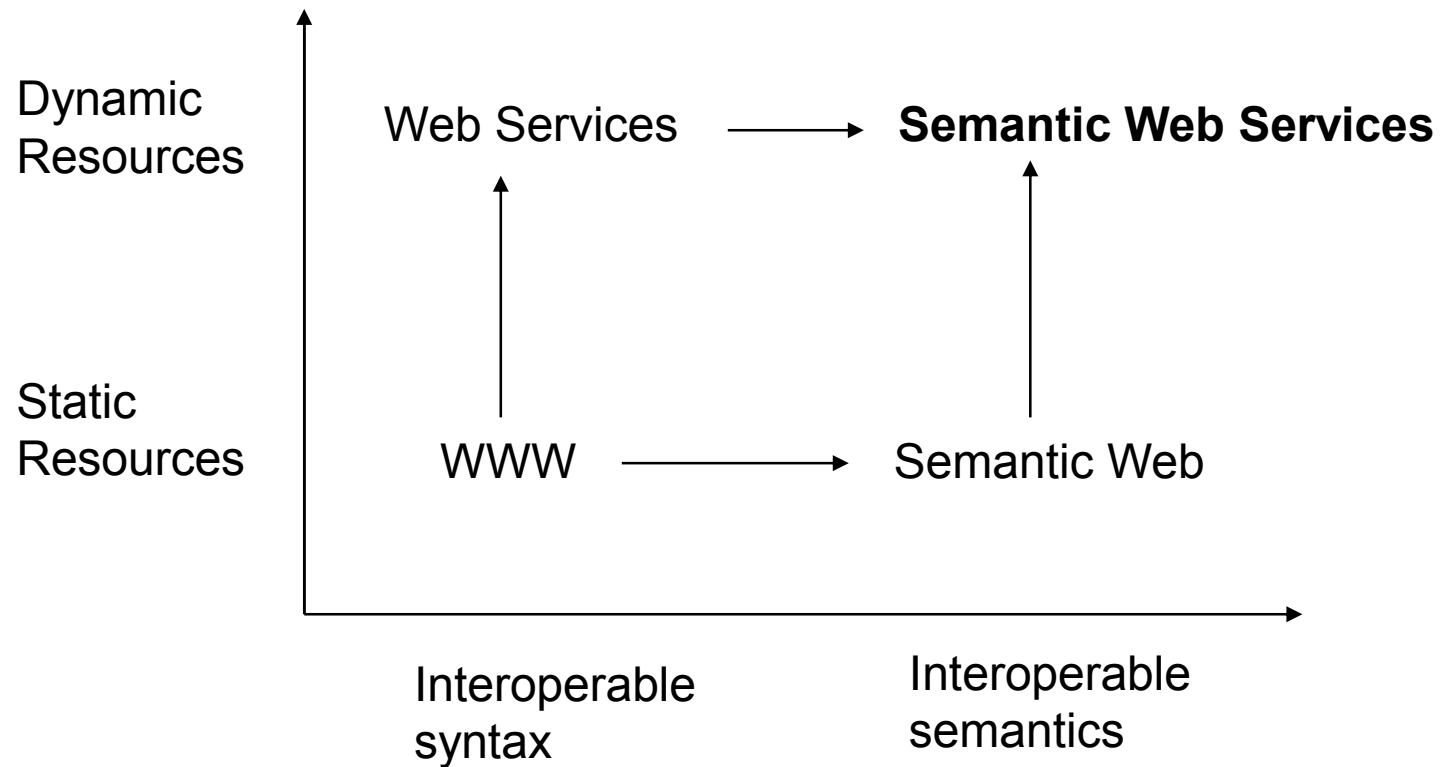
Four stages of the smart data continuum:



Stovepipe systems and the Semantic Web

- In a stovepipe system all the components are hardwired to only work together
- Information only flows in the stovepipe and cannot be shared by other systems or organizations
- E.g., the client can only communicate with specific middleware that only understands a single database with a fixed schema
- The semantic web technologies will be most effective in breaking down stovepiped database systems

Web Services and the Semantic Web



Making data smarter

- **Logical assertions:**
Connecting a subject to an object with a verb (e.g., RDF-statements)
- **Classification**
Taxonomy models, e.g. XML Topic maps
- **Formal class models**
E.g., UML- presentations
- **Rules**
An inference rule allows to derive conclusions from a set of premises, e.g. “modus ponens”

Chapter 2: The Business Cases for the Semantic Web

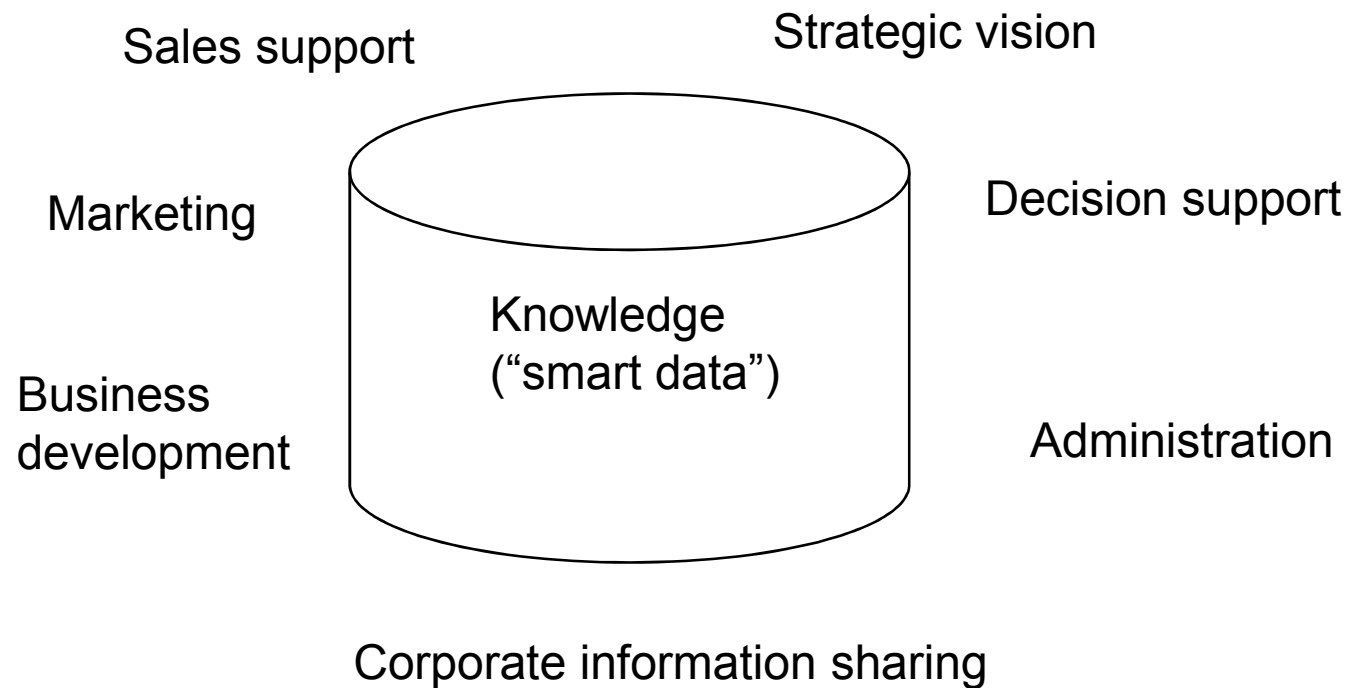


Figure. Uses of the Semantic Web in an enterprise

Chapter 3: Understanding XML and its Impact on Enterprise

- Currently the primary use of XML is for data exchange between internal and external organizations
- XML creates application-independent documents and data
- XML is a meta language; it is used for creating new language
- Any language created via the rules of XML is called an application of XML

Markup

- XML is a markup language
- A markup language is a set of words, or marks, that surround, or “tag”, a portion of a document’s content in order to attach additional meaning to the tagged content, e.g.,

```
<footnote>
```

```
  <author> Michael C. Daconta </author> <title> Java Pitfalls </title>
```

```
</footnote>
```

XML - markup

- XML – document is a hierarchical structure (a tree) comprising of *elements*
- An element consists of an opening tag, *its* content *and* a closing tag, e.g.,
`<lecturer>David Billington</lecturer>`
 - Tag names can be chosen almost freely; there are very few restrictions:
 - The first character must be a letter, an underscore, or a colon; and no name may begin with the string “XML”
 - The content may be text, or other elements, or nothing, e.g.,
`<lecturer>
 <name>David Billington</name>
 <phone>+61-7-3875 507</phone>
</lecturer>`

- If there is no content, then the element is called *empty*.

- An empty element like

- <lecturer></lecturer>

- can be abbreviated as

- <lecturer/>

- Each name / value pair attached to an element is called an *attribute*, *an element may have more than one attribute* e.g., the following element has three attributes:

<auto color="red" make = "Dodge" model = "Viper" > My car </auto>

Attributes

- An empty element is not necessarily meaningless, because it may have some properties in terms of *attributes*, e.g.,

<lecturer name =“David Billington” phone = “61-7-3875 507”/>

- The combination of elements and attributes makes XML well suited to model both relational and object-oriented data

An example of attributes for a nonempty element:

```
<order orderNo="23456" customer="John Smith" date="October 15, 2004">  
  <item itemNo="a528" quantity "1"/>  
  <item itemNo="c817 "quantity "3"/>  
</order>
```

The same information could have been written by replacing attributes by nested elements:

```
<order>  
  <orderNo>2345</orderNo>  
  <customer>John Smith</customer>  
  <date>October 15, 2004</date>  
  <item>  
    <itemNo>a528</itemNo>  
    <quantity>1</quantity>  
  </item>  
  <item>  
    <itemNo>c817</itemNo>  
    <quantity>3</quantity>  
  </item>  
</order>
```

Prologs

- An *XML-document* consists of a prolog and a number of elements
- The prolog consists of an XML-declaration and an optional reference to external structuring documents,
 - An example of *XML declaration*

```
<?xml version="1.0" encoding="UTF-16?>
```

Specifies that the document is an XML document, and defines the version and the character encoding used in the particular system (such as UTF-8, UTF-16, and ISO 8859-1)

Prologs

- It is also possible to define whether the document is self-contained, i.e., whether it does not refer external structuring documents, e.g.,

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ? >
```

A reference to external structuring documents looks like this:

```
<!DOCTYPE book SYSTEM "book.dtd">
```

Here the structuring is found in a local file called book.dtd

- If only a locally recognized name or only a URL is used, then the label SYSTEM is used.
- If one wishes to give both a local name and a URL, then the label PUBLIC should be used instead

Well – Formed and Valid XML - Documents

- *A well-formed XML document* complies with all the key W3C syntax rules of XML
 - guarantees that XML processor can parse (break into identifiable components) the document without errors
- An XML-document is well-formed if is syntactically correct. Some syntactic rules are:
 - There is only one outermost element in the document (called the *root element*)
 - Each element contains an opening and a corresponding closing tag
 - Tags may not overlap, as in
`<author><name>Lee Hong</author></name>`

Well – Formed and Valid XML - Documents

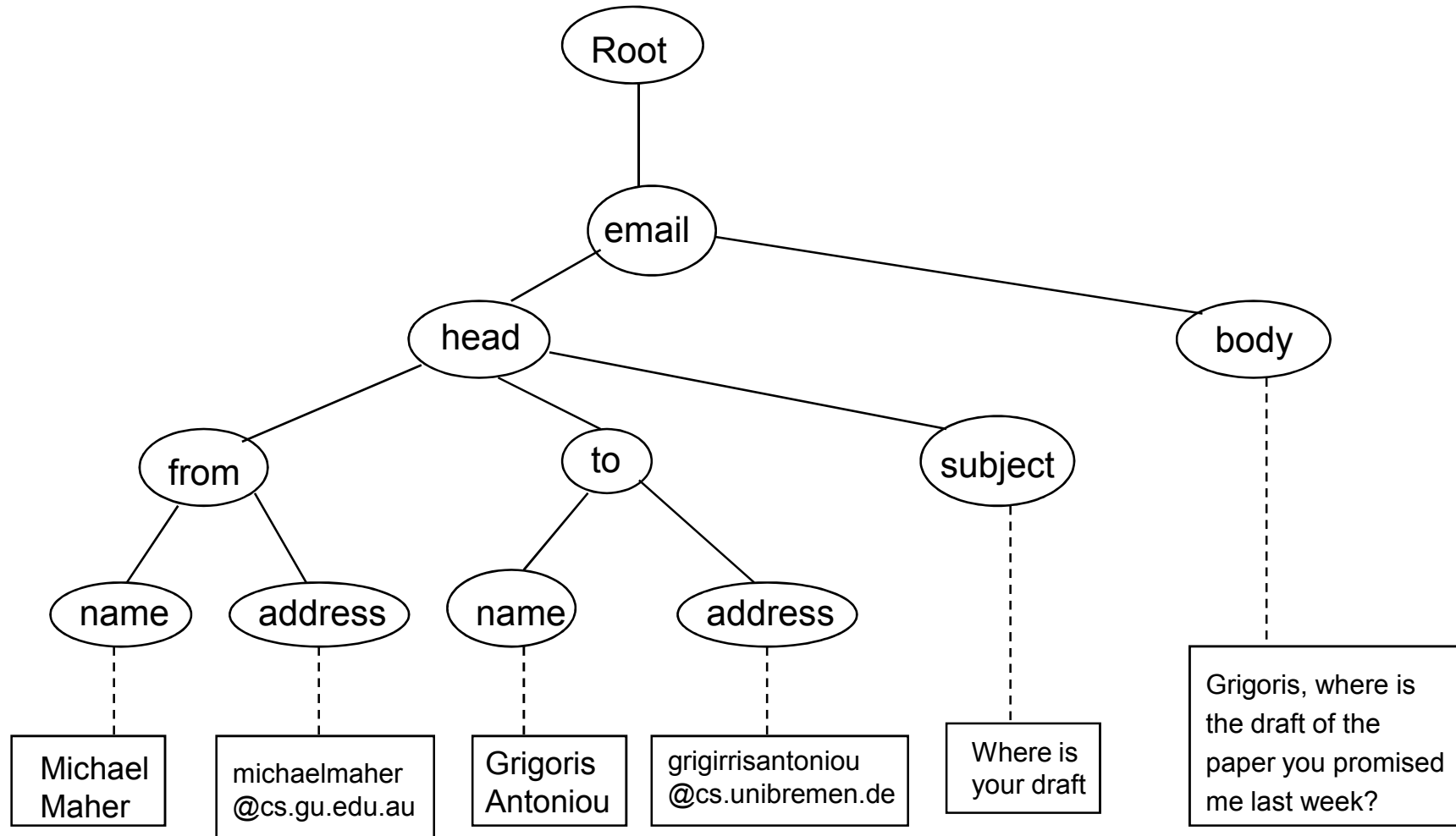
- *A valid XML document* references and satisfies a *schema*
 - *A schema* is a separate document whose purpose is to define the legal elements, attributes, and structure of an XML instance document, i.e., a schema defines a particular type or class of documents

The tree model of XML Documents

- It is possible to represent well-formed XML documents as trees; thus trees provide a formal data model for XML, e.g., the following document can be presented as a tree

```
<?xml version="1.0" encoding="UTF-16?">
  <!DOCTYPE email SYSTEM "email.dtd">
  <email>
    <head>
      <from name="Michael Maher" address ="michaelmaher@cs.gu.edu.au"/>
      <to name="Grigoris Antoniou" address ="grigoris@cs.unibremen.de"/>
      <subject>Where is your draft?</subject>
    </head>
    <body>
      Grigoris, where is the draft of the paper you promised me last week?
    </body>
  </email>
```

Tree representation of the document



DTDs

There are two ways for defining the structure of XML-documents:

- DTDs (Document Type Definition) the older and more restrictive way
 - XML-Schema which offers extended possibilities, mainly for the definition of data types
- External and internal DTDs
 - The components of a DTD can be defined in a separate file (*external DTD*) or within the XML document itself (*internal DTD*)
 - Usually it is better to use external DTDs, because their definition can be used across several documents

- Elements

- Consider the element

```
<lecturer>  
  <name>David Billington</name>  
  <phone>+61-7-3875 507</phone>  
</lecturer>
```

- A DTD for this element type looks like this:

```
<!ELEMENT lecturer (name, phone)>  
<!ELEMENT name (#PCDATA)>  
<!ELEMENT phone (#PCDATA)>
```

- In DTDs #PCDATA is the only atomic type of elements
- We can express that a lecturer element contains either a name element or a phone element as follows:

```
<!ELEMENT lecturer (name | phone)>
```

- Attribute

- Consider the element

```
<order orderNo="23456" customer="John Smith" date="October 15, 2004">  
  <item itemNo="a528" quantity "1"/>  
  <item itemNo="c817 "quantity "3"/>  
</order>
```

A DTD for it looks like this:

```
<!ELEMENT order (item+)>  
<!ATTLIST order  
  orderNo ID #REQUIRED  
  customer CDATA #REQUIRED  
  date CDATA #REQUIRED>  
<!ELEMENT item EMPTY>  
<!ATTLIST item  
  itemNo ID #REQUIRED  
  quantity CDATA #REQUIRED  
  comments CDATA #IMPLIED>
```

NOTE. Compared to the previous example now the *item* element type is defined to be empty

- Cardinality operators
 - ? : appears zero times or once
 - * : appears zero or more times
 - + : appears one or more times
 - No cardinality operator means exactly one
 - CDATA, a string (a sequence of characters)

Example: DTD for the email document

```
<!ELEMENT          email (head, body)>
<!ELEMENT          head (from, to+, cc*, subject)>
<!ELEMENT          from EMPTY>
<!ATTLIST         from
  name            CDATA          #IMPLIED
  address         CDATA          #REQUIRED>
<!ELEMENT          to EMPTY>
<!ATTLIST         to
  name            CDATA          #IMPLIED
  address         CDATA          #REQUIRED>
<!ELEMENT          cc EMPTY>
<!ATTLIST         cc
  name            CDATA          #IMPLIED
  address         CDATA          #REQUIRED>
<!ELEMENT          subject      (#PCDATA)>
<!ELEMENT          body        (text, attachment*)>
<!ELEMENT          text        (#PCDATA)
<!ELEMENT          attachment  EMPTY>
<!ATTLIST         attachment  encoding (mime | binhex "mime" file CDATA
#REQUIRED>
```

Some comments for the email DTD

- A head element contains a *from* element, at least one *to element*, zero or more *cc* elements, and a *subject* element, in the order
- In *from*, *to* and *cc* elements the name attribute is not required; the address attribute on the other hand is always required.
- A *body* element contains a *text* element, possibly followed by a number of *attachment* elements
- The encoding attribute of an *attachment* element must have either the value “mime” or “binhex”, the former being the default value.
- #REQUIRED. The Attribute must appear in every occurrence of the element type in the XML-document.
- #IMPLIED. The appearance of the attribute is optional

NOTE. A DTD can be interpreted as an Extended Backus-Naur Form (EBNF).

For example, the declaration

```
<!ELEMENT email (head, body)>
```

is equivalent to the rule

```
email :: head body
```

which means that e-mail consists of head followed by a body.

Data Modeling Concepts

XML	Object-oriented	Relational
Element	Class	Entity
Attribute	Data member	Relation

XML-Schema

- XML Schema offers a significantly richer language than DTD for defining the structure of XML-documents
- One of its characteristics is that its syntax is based on XML itself
 - This design decision allows significant reuse of technology
- XML-Schema allows one to define new types by extending or restricting already existing ones
- XML-Schema provides a sophisticated set of data types that can be used in XML documents (DTDs were limited to strings only)

XML-Schema

- XML – Schema is analogous to a database schema, which defines the column names and data types in database tables
- The roles of the XML-Schema:
 - Template for a form generator to generate instances of a document type
 - Validator to ensure the accuracy of documents
- XML-Schema defines element types, attribute types, and the composition of both into composite types, called complex types

XML-Schema

- An XML Schema is an element with an opening tag like

```
<XSD:schema
```

```
  xmlns:xsd=http://www.w3.org/2000/10/XMLSchema
```

```
  version="1.0">
```

- The element uses the schema of XML Schema found at W3C Web site. It is the “foundation” on which new schemas can be built
- The prefix xsd denotes the namespace of that schema. If the prefix is omitted in the xmlns attribute, then we are using elements from this namespace by default:

```
<schema
```

```
  xmlns=http://www.org/2000/10/XMLSchema version="1.0">
```

XML-Schema

- An XML Schema uses XML syntax to declare a set of simple and complex type declarations
 - A type is a named template that can hold one or more values
 - Simple types hold one value while complex types are composed of multiple simple types
 - An example of a simple type:
`<xsd:element name = "author" type "xsd:string" />`
(note: "xsd:string" is a built-in data type)
Enables instance elements like:
`<author> Mike Daconta </author>`

XML Schema

- A *complex type* is an element that either contains other elements or has attached attributes, e.g., (attached attributes):

```
<xsd: element name = " book">  
  <xsd: complexType>  
    <xsd: attribute name = "title" type = "xsd: string" />  
    <xsd: attribute name = "pages" type = "xsd: string" />  
  </xsd: complexType>  
</xsd: element>
```

An example of the book element would look like:

```
<book title = "More Java Pitfalls" pages = "453" />
```

XML Schema

- XML-Schema “product” has attributes and child elements:

```
<xsd: element name = “product”>
  <xsd: complexType>
    <xsd: sequence>
      <xsd: element name=“description” type=“xsd:string”
        minoccurs=“0” maxoccurs=“1” />
      <xsd: element name=“category” type=“xsd:string”
        minoccurs=“1” maxOccurs=“unbounded” />
    </xsd:sequence>
    <xsd: attribute name= “id” type=“xsd:ID” />
    <xsd: attribute name=“title” type=“xsd:string” />
    <xsd: attribute name=“price” type=“xsd:decimal” />
  </xsd: complexType>
</xsd: element>
```

XML Schema

- An XML-instance of the product element:

```
<product id =“PO1” title=“Wonder Teddy” price=“49.99”>  
  <description>  
    The best selling teddy bear of the year  
  </description>  
  <category> toys </category>  
  <category> stuffed animals </category>  
</product>
```

XML Schema

- An other XML-instance of the product element:

```
<product id="P02" title="RC Racer" price="89.99">  
  <category> toys </category>  
  <category> electronic </category>  
  <category> radio-controlled </category>  
</product>
```


Data Types

- There is a variety of *built-in datatypes* including:
 - Numerical data types, including *integer, Short, Byte, Long, Float, Decimal*
 - String data types, including, *string, ID, IDREF, CDATA, Language*
 - Date and time data types, including, *Time, Date, Month, Year*
- *Complex types are defined from already existing data types by defining some attributes (if any) and using*
 - Sequence, a sequence of existing data type elements, the appearance of which in a predefined order is important
 - All, a collection of elements that must appear, but the order of which is not important
 - Choice, a collection of elements, of which one will be chosen

Data Types: example

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="lastname" type="string"/>
  </sequence>
  <attribute name="title" type="string" use="optional"/>
</complexType>
```

The meaning is that an element in an XML document that is declared to be of type *lecturerType* may have *title* attribute, any number of *firstname* elements, and exactly one *lastname* element.

Data Type Extension

- Existing data type can be extended by new elements or attributes
- As an example, we extend the *lecturer* data type

```
<complexType name="extendedLecturerType">  
  <extension base="lecturerType">  
    <sequence>  
      <element name="email" type="string"  
        minoccurrence="0" maxoccurrence="1"/>  
    </sequence>  
    <attribute name="rank" type="string use="required"/>  
  </extension>  
</complexType>
```

Data Type Extension

The resulting data type looks like this:

```
<complexType name="extendedlecturerType">
  <sequence>
    <element name="firstname" type="string"
      minoccurs="0" maxoccurs="unbounded"/>
    <element name="lastname" type="string"/>
    <element name="email" type="string"
      minoccurs="0" maxoccurs="1"/>
  </sequence>
  <attribute name="title" type="string" use="optional"/>
  <attribute name="rank" type="string" use="required"/>
</complexType>
```

Data Type Restriction

- An existing data type may also be restricted by adding constraints on certain values
 - E.g., new *type* and *use* attributes may be added or the numerical constraints of *minOccurs* and *maxOccurs* tightened
 - As an example, we restrict the lecturer data type as follows (tightened constraints are shown in **boldface**):

```
<complexType name="RestrictedLecturerType">  
  <restriction base="lecturerType">  
    <sequence>  
      <element name="firstname" type="string"  
        minOccurs="1" maxOccurs="2"/>  
      <element name="lastname" type="string"/>  
    </sequence>  
    <attribute name="title" type="string" use="required"/>  
  </restriction>  
</complexType>
```

XML-namespaces

- *Namespaces* is a mechanism for creating globally unique names for the elements and attributes of the markup language
- Namespaces are implemented by requiring every XML name to consists of two parts: a prefix and a local part, e.g., `<xsd: integer>`

here the local part is “integer” and the prefix is an abbreviation for the actual namespace in the namespace declaration. The actual namespace is a unique Uniform Resource Identifier.

A sample namespace declaration:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
```

XML-namespaces

- There are two ways to apply a namespace to a document:

attach the prefix to each element and attribute in the document, or declare a default namespace for the document, e.g.,

```
<html xmlns=http://www.w3.org/1999/xhtml>  
<head> <title> Default namespace test </title> </head>  
<body> Go Semantic Web ! </body>  
</html>
```

XML-namespaces: Example

- Consider an (imaginary) joint venture of an Australian university, say Griffith University, and an American University, say University of Kentucky, to present a unified view for online students
- Each university uses its own terminology and there are differences; e.g., lecturers in the United States are not considered regular faculty, whereas in Australia they are (in fact, they correspond to assistant professors in the United States)
- The following example shows how disambiguation can be achieved


```
<?xml version="1.0" encoding="UTF-16"?>
<vu : instructors
  xmlns : vu="http://www.vu.com/empDTD"
  xmlns : gu="http://www.gu.au/empDTD"
  xmlns : uky="http://www.uky.edu/empDTD" >
  <uky : faculty
    uky : title="assistant professor"
    uky : name="John Smith"
    uky : department="Computer Science"/>
  <gu : academicStaff
    gu : title="lecturer"
    gu : name="Mate Jones"
    gu : school="Information Technology"/>
</vu : instructors>
```

If a prefix is not defined, then the location is used by default. So, for example the previous example is equivalent to the following document (differences are shown in **boldface**)

```
<?xml version="1.0" encoding="UTF-16"?>
<vu : instructors
  xmlns : vu="http://www.vu.com/empDTD"
  xmlns="http://www.gu.au/empDTD"
  xmlns : vu="http://www.uky.edu/empDTD" >
  <uky : faculty
    uky : title="assistant professor"
    uky : name="John Smith"
    uky : department="Computer Science"/>
  <academicStaff
    title="lecturer"
    name="Mate Jones"
    school="Information Technology"/>
</vu : instructors>
```

Example: XML-Schema for the email document

```
<schema xmlns=http://www.org/2000/10/XMLSchema version="1.0">
  <element name="email" type="emailtype"/>
  <complexType name="emailType">
    <sequence>
      <element name="head" type="headType"/>
      <element name="body" type="bodyType"/>
    </sequence>
  </complexType>
  <complexType name="headType">
    <sequence>
      <element name="from" type="nameAddress"/>
      <element name="to" type="nameAddress"
        minOccurs="1" maxoccurs="unbounded"/>
      <element name="cc" type="nameAddress"
        minOccurs="0" maxoccurs="unbounded"/>
      <element name="subject" type="string"/>
    </sequence>
  </complexType>
```

```
<complexType name="nameAddress">  
  <attribute name="name" type="string" use="optional"/>  
  <attribute name="address" type="string" use="required"/>  
</complexType>
```

```
<complexType name="bodyType">
  <sequence>
    <element name="text" type="string"/>
    <element name="attachment" minOccurs="0"
      maxOccurs="unbounded"/>
      <complexType>
        <attribute name="encoding" use="default" value="mime">
          <simpleType>
            <restriction base="string">
              <enumeration value="mime"/>
              <enumeration value="binhex"/>
            </restriction>
          </simpleType>
        </attribute>
        </attribute name="file" type="string" use="required"/>
      </complexType>
    </element>
  </sequence>
</complexType>
```

Uniform Resource Identifier (URI)

- URI is a standard syntax for strings that identify a resource
- Informally, URI is a generic term for addresses and names of objects (or resources) on the WWW.
- A resource is any physical or abstract thing that has an identity
- There are two types of URI:s:
 - Uniform Resource Locator (URL) identifies a resource by how it is accessed, e.g., <http://www.example.com/stuff/index.html> identifies a HTML page on a server
 - Uniform Resource Names (URNs) creates a unique and persistent name for a resource either in the “urn” namespace or another registered namespace.

Document Object Model (DOM)

- DOM is a data model, using objects, to represent and manipulate an XML or HTML documents
- Unlike XML instances and XML schemas, which reside in files on disks, the DOM is an in-memory representation of a document.
- In particular, DOM is an application interface (API) for programmatic access and manipulation of XML and HTML

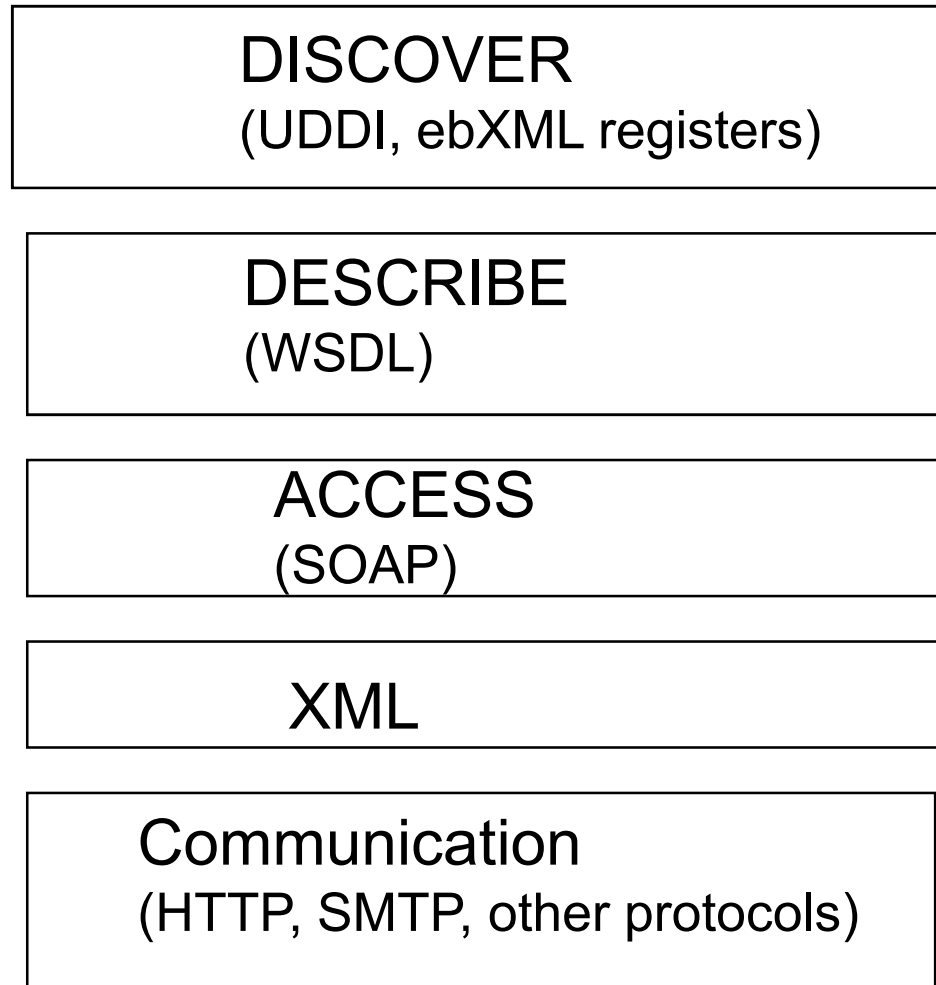
Semantic Levels of Modeling

Level 3 (Worlds)	Ontologies (rules and logic)
Level 2 (Knowledge about things)	RDF, taxonomies
Level 1 (Things)	XML Schema, conceptual models

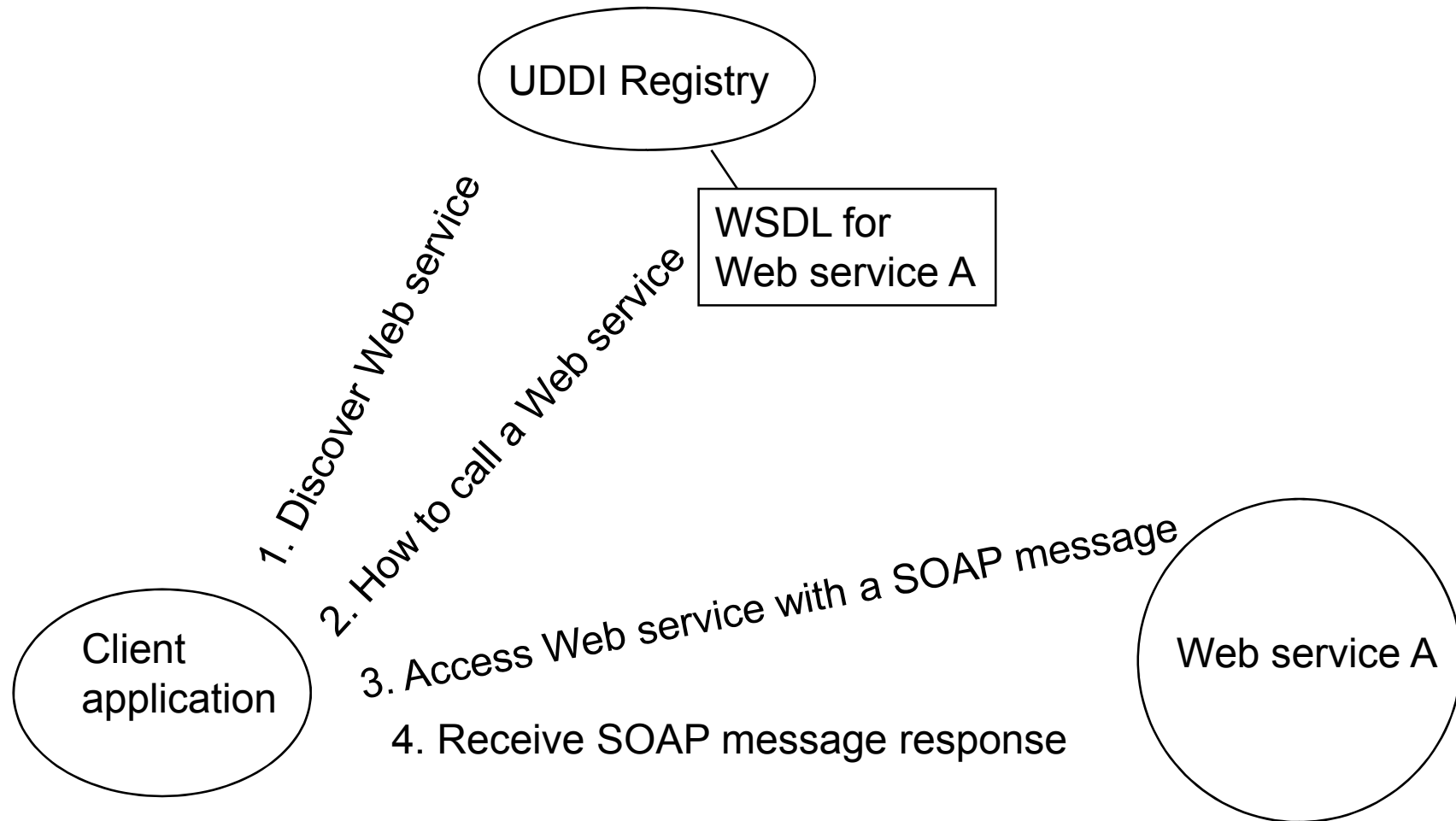
Chapter 4: Understanding Web Services

- Web services provide interoperability solutions, making application integration and transacting business easier
- Web services are software applications that can be discovered, described and accessed based on XML and standard Web protocols over intranets, extranets, and the Internet

The basic layers of Web services



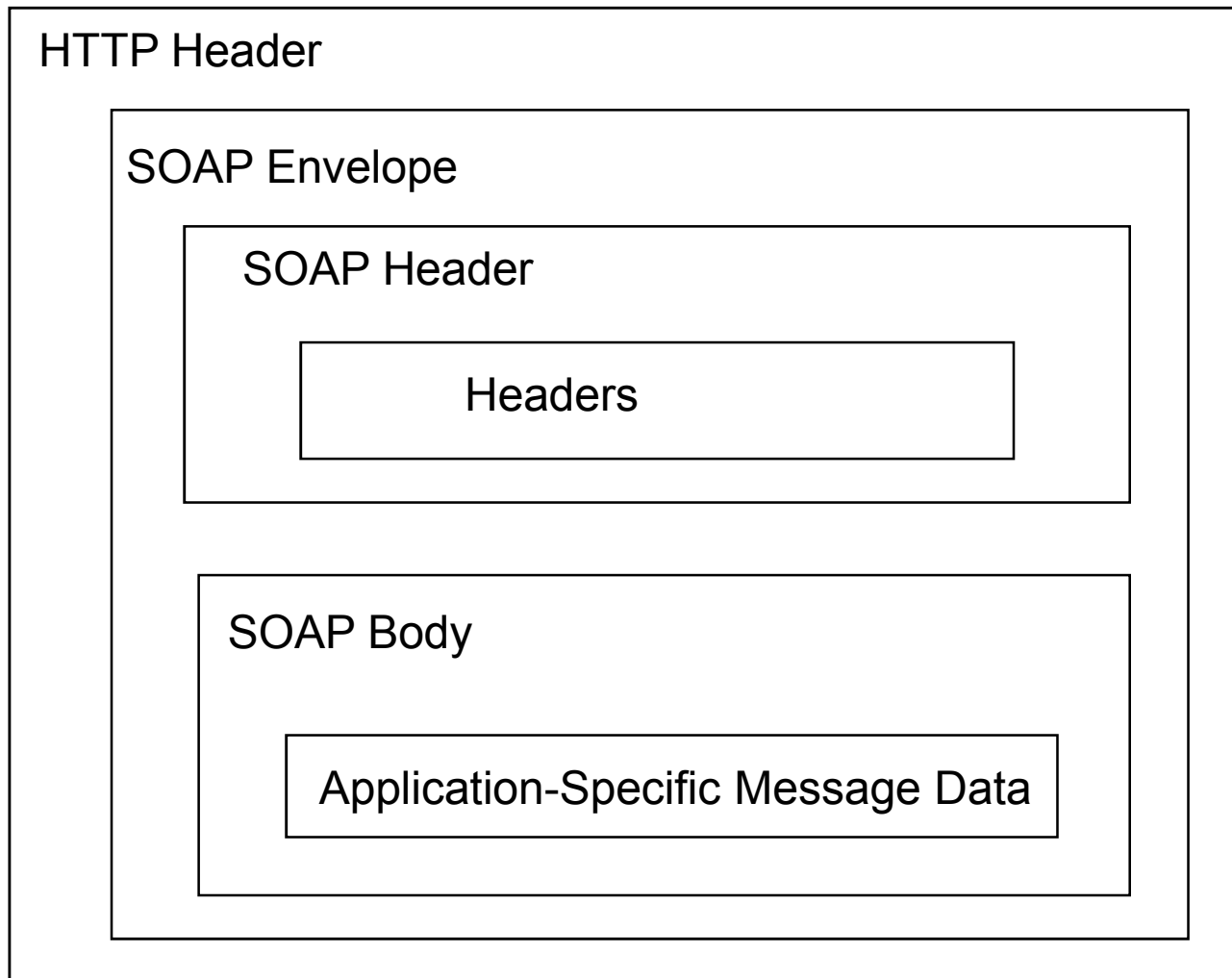
A common scenario of Web service use



SOAP

- SOAP (“Simple Object Access Protocol”) is the envelope syntax for sending and receiving XML-messages with Web services
- An application sends a SOAP request to a Web service, and the Web service returns the response.
- SOAP can potentially be used in combination with a variety of other protocols, but in practice, it is used with HTTP

The structure of a SOAP message



An example: SOAP message for getting the last trade price of “DIS” ticker symbol

```
<SOAP-ENV: Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" >
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI" >
      <symbol> DIS </symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV: Body>
</SOAP-ENV: Envelope>
```

The SOAP response for the example stock price request:

```
<SOAP-ENV: Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns: m="Some-URI" >
      <Price> 34.5 </Price>
    </m:GetLastTradePrice>
  </SOAP-ENV: Body>
</SOAP-ENV: Envelope>
```

Web Service Definition Language (WSDL)

- WSDL is a language for describing the communication details and the application-specific messages that can be sent in SOAP.
- To know how to send messages to a particular Web service, an application can look at the WSDL and dynamically construct SOAP messages.

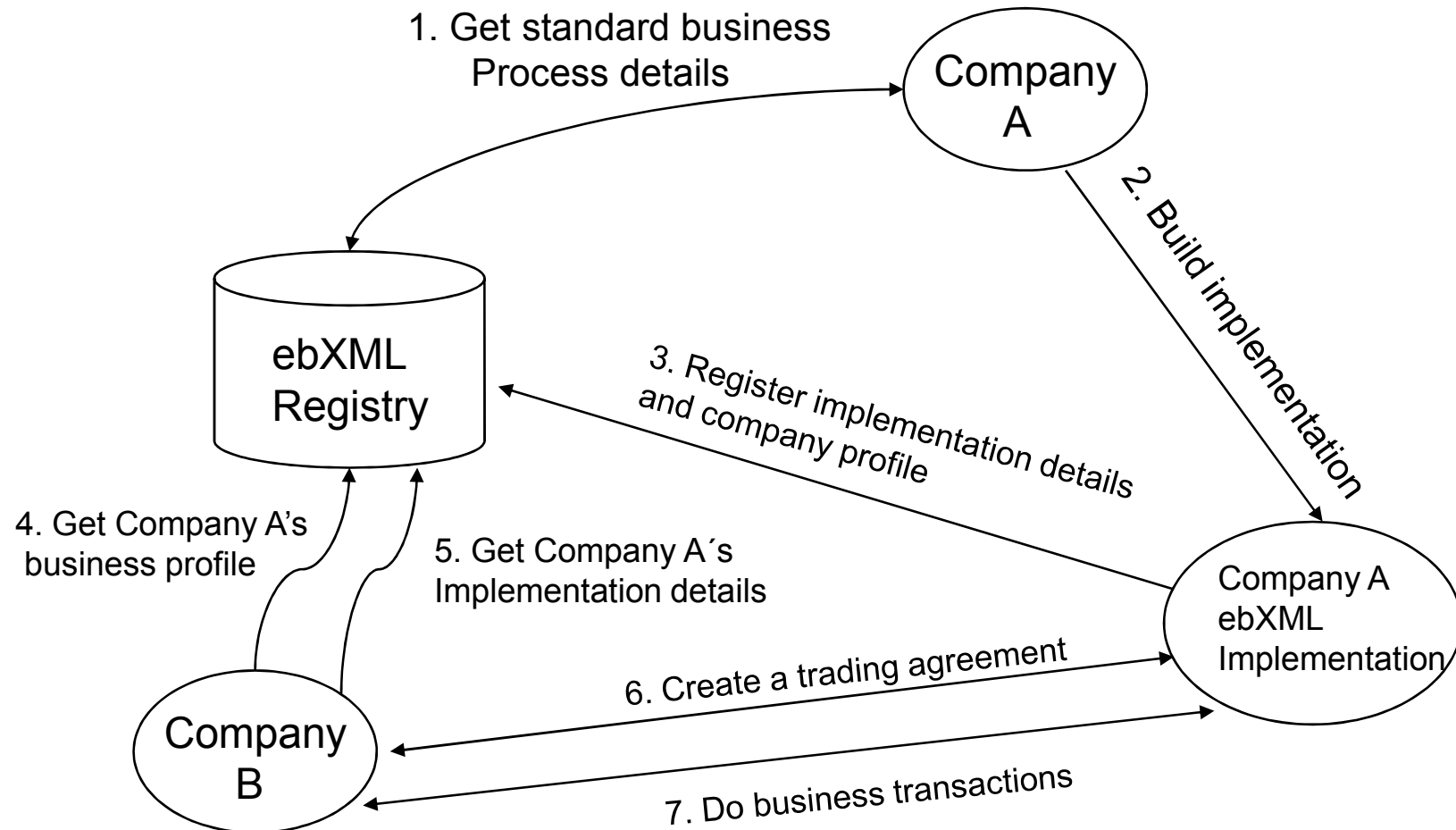
Universal Description, Discovery, and Integration (UDDI)

- Organizations can register public information about their Web services and types of services with UDDI, and applications can view this information
- UDDI register consists of three components:
 - White pages of company contact information,
 - Yellow pages that categorize business by standard taxonomies, and
 - Green pages that document the technical information about services that are exposed
- UDDI can also be used as internal (private) registers

ebXML Registries

- ebXML standard is created by OASIS to link traditional data exchanges to business applications to enable intelligent business processes using XML
- ebXML provides a common way for business to quickly and dynamically perform business transactions based on common business practices
- Information that can be described and discovered in an ebXML architectures include the following:
 - Business processes and components described in XML
 - Capabilities of a trading partner
 - Trading partner agreements between companies

An ebXML architecture in use



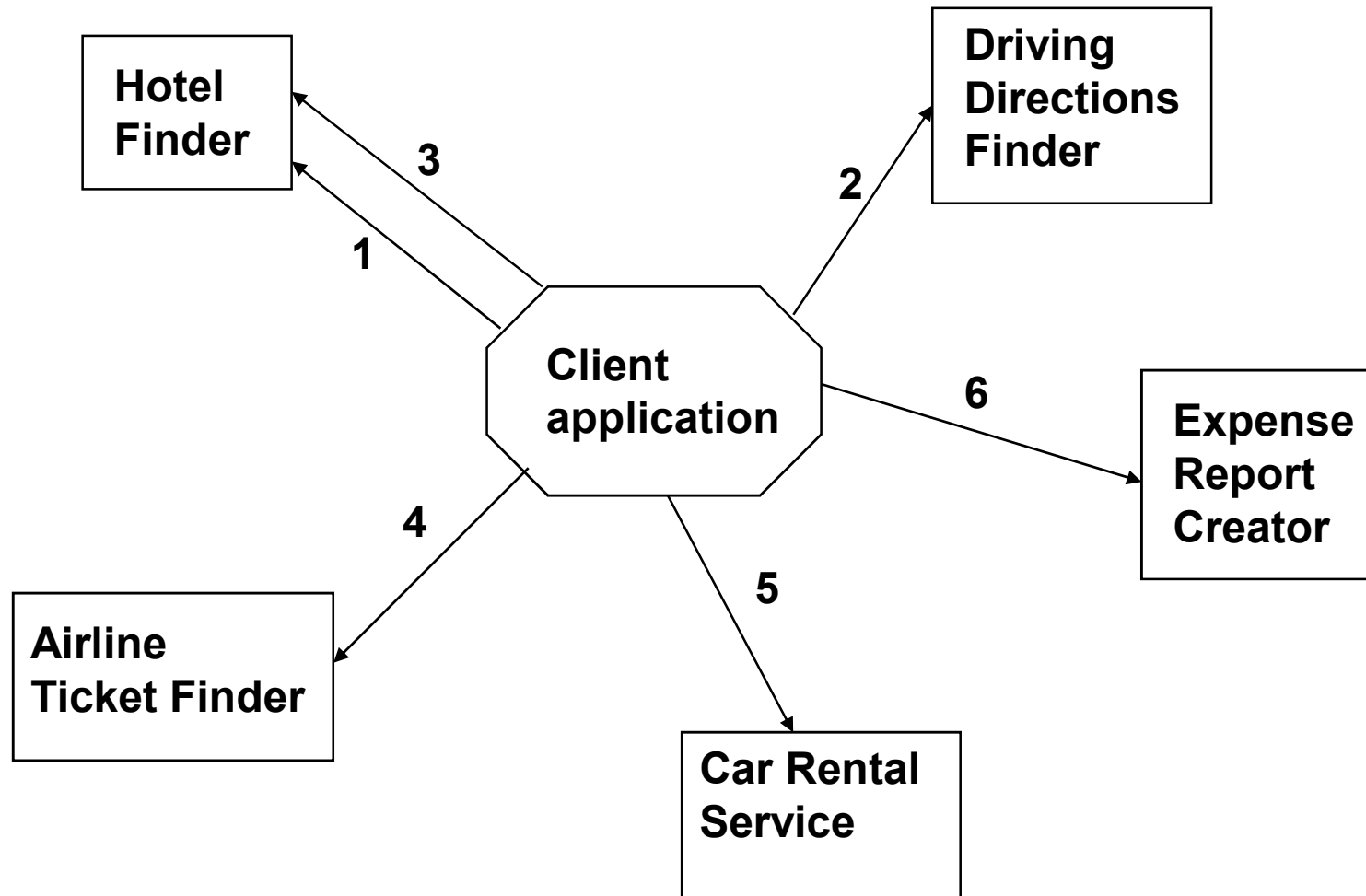
Orchestrating Web Services

- *Orchestration* is the process of combining simple Web services to create complex, sequence-driven tasks, called *Web service choreography*, or *Web workflow*
- *Web workflow* involves creating business logic to maintain conversation between multiple Web services.
- Orchestration can occur between
 - an application and multiple Web services, or
 - multiple Web services can be chained in to a workflow, so that they can communicate with one another

Web workflow example

- Hotel finder Web service
 - provides the ability to search for a hotel in a given city, list room rates, check room availability, list hotel amenities, and make room reservations
- Driving directions finder
 - Gives driving directions and distance information between two addresses
- Airline ticket booker
 - Searches for flights between two cities in a certain timeframe, list all available flights and their prices, and provides the capability to make flight reservations
- Car rental Web service
 - Provides the capability to search for available cars on a certain date, lists rental rates, and allows an application to make a reservation
- Expense report creator
 - Creates automatically expense reports, based on the sent expense information

Example continues: Orchestration between an application and the Web services



The steps of the example

1. The client application send a message to the hotel finder Web service in order to look for the name, address, and the rates of hotels (e.g., with nonsmoking rooms, local gyms, and rates below \$150\$ a night) available in the Wailea, Maui, area during the duration of the trip
2. The client application send a message to the driving directions finder Web service. For the addresses returned in Step 1, the client application requests the distance to Big Makena Beach. Based on the distance returned for the requests to this Web service, the client application finds the four closest hotels.
3. The client application requests the user to make a choice, and then the client application sends an other message to the hotel finder to make the reservation
4. Based on the user's frequent flyer information, e.g., on Party Airlines, and the date of the trip to Maui, the client application send a message to the airline ticket booker Web service, requesting the cheapest ticket

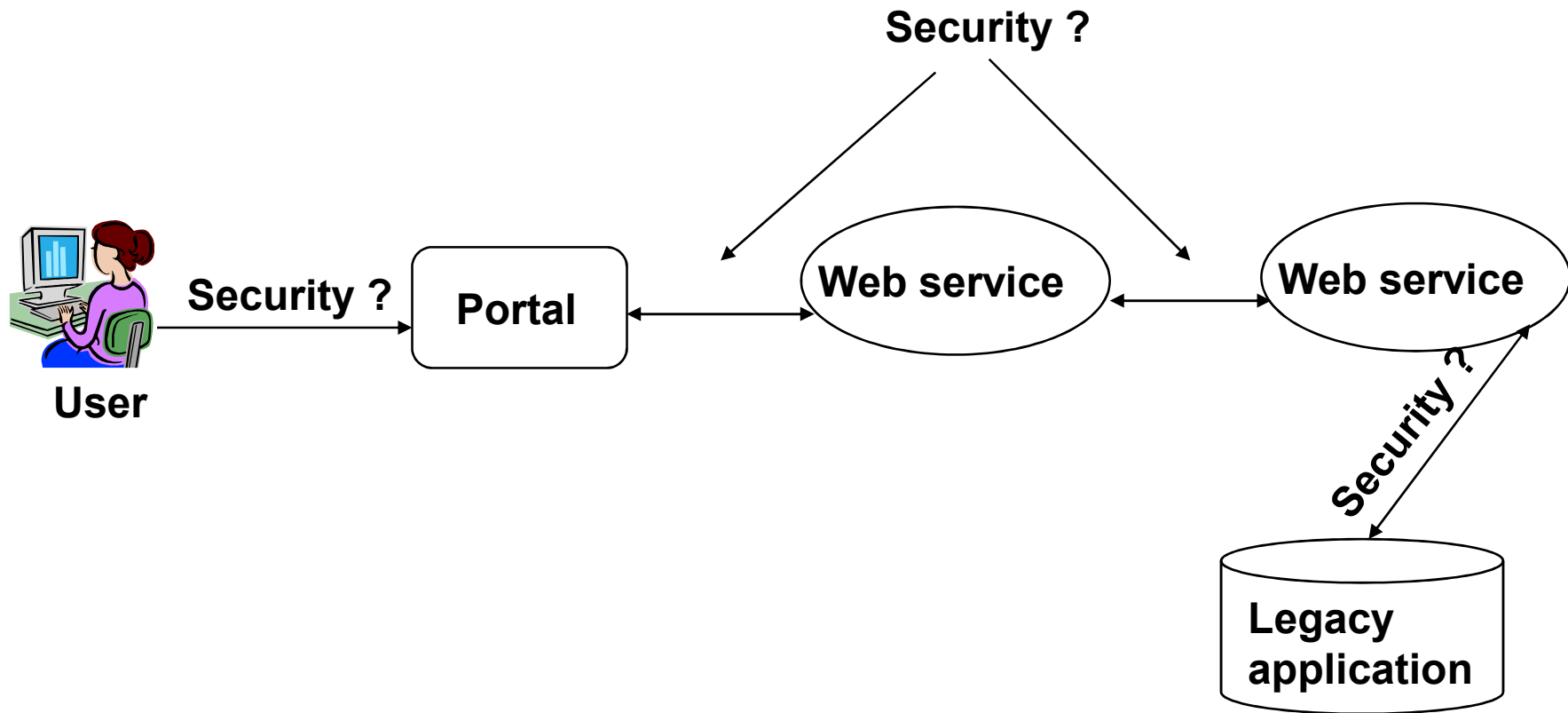
The steps of the example, continues ...

- 5 The client application send a message to the car rental Web service and requests the cheapest rentals. In the case of multiple choices the client application prompts the user to make a choice.
 - 6 Sending all necessary receipt information found in Step 1 to 5, the client application requested an expense report generated from the expense report creator Web service. The client application then emails the resulting expense report, in the corporate format, to the end user.
- **Note:** the above example may be processes either in *Intranet*, meaning that the Web services are implemented in Intranet and so the client application knows all the Web service calls in advance, or in *Internet*, meaning that the client application may discover the available services via UDDI and download the WSDL for creating the SOAP for querying the services, and dynamically create those messages on the fly. This approach requires the utilization of ontologies.

Security of Web services

- One of the biggest concerns in the deployment of Web services is security
- Today, most internal Web service architectures (Intranet and to some extent extranets), security issues can be minimized
- Internal EAI (Enterprise Application Integration) projects are the first areas of major Web service rollouts

Security at different points



Security related aspects

- Authentication
 - Mutual authentication means proving the identity of both parties involved in communication
 - Message origin authentication is used to make certain that the message was sent by the expected sender
- Authorization
 - Once a user's identity is validated, it is important to know what the user has permission to do
 - Authorization means determining a user's permissions
- Single sign-on (SSO)
 - Mechanism that allows user to authenticate only once to her client, so that no new authentication for other web services and server applications is not needed

Security related aspects, continues ...

- Confidentiality
 - Keeping confidential information secret in transmission
 - Usually satisfied by encryption
- Integrity
 - Validating message's integrity means using techniques that prove that data has not been altered in transit
 - Techniques such as hash codes are used for ensuring integrity
- Nonrepudiation
 - The process of proving legally that a user has performed a transaction is called nonrepudiation

Chapter 5: Understanding Resource Description Framework (RDF)

Motivation

- XML is a universal meta language for defining markup; it does not provide any means of talking about the *semantics* (meaning) of data
 - E.g., there is no intended meaning associated with the nesting of tags
 - To illustrate this, assume that we want to express the following fact

David Billington is a lecturer of Discrete Mathematics

There are various ways of representing this sentence in XML:

```
<course name="Discrete Mathematics">  
  <lecturer>David Billington</lecturer>  
</course>
```

```
<lecturer name="David Billington">  
  <teaches>Discrete Mathematics</teaches>  
</lecturer>
```

```
<teachingOffering>  
  <lecturer>David Billington</lecturer>  
  <course>Discrete Mathematics</course>  
</teachingOffering>
```

Note. The first two formalizations include essentially an opposite nesting although they represent the same information. So there is no standard way of assigning meaning to tag nesting.

RDF continues ..

- RDF (Resource Description Framework) is essentially a *data-model*
- Its basic block is object-attribute-value triple (subject- predicate-object triple according to RDF-terminology), called a *statement*,
 - E.g., “*David Billington is a lecturer of Discrete Mathematics*”
is such a statement
- An abstract data needs a concrete syntax in order to be represented and transmitted, and RDF has been given a syntax in XML
 - As a result RDF inherits the benefits associated with XML
 - However, other syntactic representations, not based on XML, are also possible

- RDF is domain-independent in that no assumptions about a particular domain of use are made
 - It is up to users to define their own terminology in a schema language RDFS (RDF Schema)
- **NOTE.** The name RDF Schema is not logical in the sense that it suggests that RDF Schema has a similar relation to RDF as XML Schema has to XML, but in fact this is not the case:
 - XML Schema constraints the *structure* of XML-documents, whereas RDF Schema defines the *vocabulary* used in RDF data models
- In RDFS we can define
 - the vocabulary,
 - specify which properties apply to which kinds of objects and
 - what values they can take, and
 - describe the relationships between the objects

- For example using RDF we can write:

Lecturer is a subclass of academic staff member

- An important point here is that there is an intended meaning associated with “is a subclass of”
 - It is not up to the application to interpret this term; its intended meaning must be respected by all RDF processing software
- Through fixing the semantics of certain ingredients, RDF/RDFS enables us to model particular domains

- For example, consider the following XML elements:

```
<academicStaffMember>Grigoris Antoniou</academicStaffMember>
```

```
<professor>Michael Maher</professor>
```

```
<course name=“Discrete Mathematics”>
```

```
  <isTaughtBy>David Billington</isTaughtBy>
```

```
</course>
```

- Suppose now that we want to collect all academic staff members
 - A path expression in XPath might be
`//academicStaffMember`
 - Its result is only Grigoris Antoniou. While it is correct from the XML viewpoint, the result is semantically unsatisfactory; human reader would have also included Michael Maher and David Billington in the result because
 - All professors are academic staff members, i.,e., *professor* is a subclass of *academicStaffMember*
 - Courses are only taught by academic staff members
 - This kind of information makes use of the *semantic model* of the particular domain, and cannot be represented in XML or in RDF but is typical of knowledge written in RDF Schema
 - Thus RDFS makes semantic information machine-accessible, in accordance with the Semantic Web vision

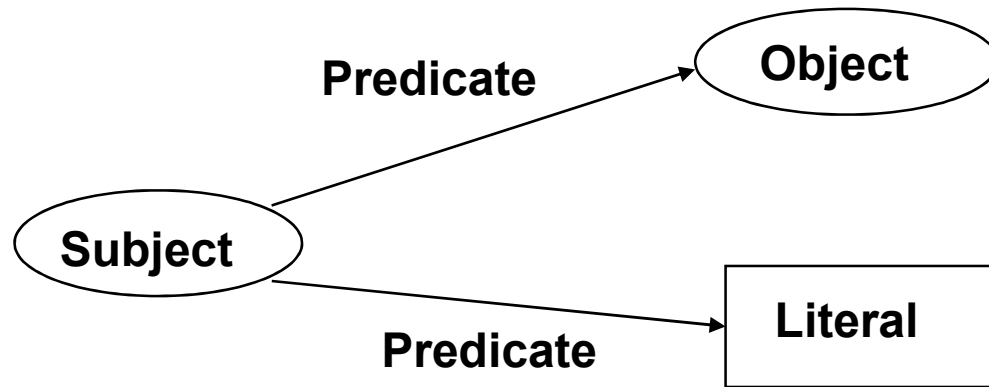
RDF continues ...


- RDF is an XML-based language to describe *resources*
- A *resource* is an electronic file available via the Uniform Resource Locator (URL)
- While XML documents attach meta data to parts of a document, **one** use of RDF is to create meta data about the document as a standalone entity, i.e., instead of marking up the internals of a document, RDF captures meta data about the “externals” of a document, like the author, creation date and type
- A particularly good use of RDF is to describe resources, which are “opaque” like images or audio files

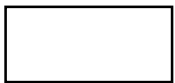
RDF continues ...

- An RDF document contains one or more “descriptions” of resources
- A *description* is a set of *statements* about a source
- An *rdf:about* attribute refers to the resource being described
- The RDF model is called a “triple” as it has three parts: subject, predicate and object

The RDF triple



 = URL

 = Literal

 = Property or Association

The elements of an RDF triple:

- Subject:
 - In grammar, the noun or noun phrase that is the doer of the action
 - E.g., in the sentence “ The company sells batteries” the subject is “the company”
 - In RDF the subject is the resource that is being described, and so we want “the company” to be an unique concept which have an URI (e.g., “<http://www.business.org/ontology/#company>”)

The elements of an RDF triple:

- Predicate:
 - In grammar the part of a sentence that modifies the subject and includes the verb phrase
 - E.g., in the sentence “ The company sells batteries” the predicate is the phrase “sells batteries”, so the predicate tells something about the subject
 - In logic, a predicate is a function from individuals (a particular type of subject) to truth-values
 - In RDF, a predicate is a relation between the subject and the object, so in RDF we would define a unique URI for the concept “sells” like <http://www.business.org/ontology/#sells>

The elements of an RDF triple:

- Object:
 - In grammar, a noun that is acted upon by the verb
 - E.g., in the sentence “ The company sells batteries” the object is the noun “batteries”
 - In logic, an object is acted upon by the predicate
 - In RDF, an object is either a resource referred to by the predicate or a literal value, so in RDF we would define a unique URI for the concept ”batteries” like <http://www.business.org/ontology/#batteries>

Capturing knowledge with RDF

- The expression of contents can be done at many ways, e.g., :
 - As natural language sentences,
 - In a simple triple notation called N3
 - In RDF/XML serialization format
 - As a graph of the triples

Expressing contents as natural language sentences

- Following the linguistic model of subject, predicate and object, we express three English statements:

Buddy Belden owns a business

The business has a Web site accessible at <http://www.c2i2.com/-budstv>.

Buddy is the father of Lynne

Expressing contents by N3 notation

- By extracting the relevant subject, predicate, and object we get the N3 notation:

<#Buddy> <#owns> <#business>

<#business> <#has-website> <http://www.c2i2.com/budstv>

<#Buddy> <#father-of> <#Lynne>

where # sign means the URI of the concept (a more accurate expression is one where # sign is replaced by an absolute URI like “http://www.c2i2com/buddy/ontology” as a formal namespace)

In N3 this can be done with a prefix tag like

@prefix bt: < http://www.c2i2com/buddy/ontology >

Using this prefix the first sentence would be:

<bt: Buddy> <bt:owns> <bt:business>

Tools are available to automatically convert the N3 notation into RDF/XML format

Example: RDF/XML generated from N3

```
<rdf:RDF
  xmlns:RDFNsId1='#'
  xmlns:rdf='http://www.w3org/1999/02/22-rdf-syntax-ns#'>

  <rdf:Description rdf:about='#Buddy'>
    <RDFNsId1:owns>
      <rdf:Description rdf:about='#business'>
        <RDFNsId1:has-website
          rdf:resource='http://www.c2i2.com/-budstv' />
        </rdf: Description>
      </RDFNsId1:owns>
      <RDFNsID1:father-of rdf:resources='#Lynne' />
    </rdf:Description>
  </rdf:RDF>
```

Expressing the domain of university courses and lectures at Griffith University by RDF/XML

```
<!DOCTYPE owl [<!ENTITY xsd "http://www.org/2001/XMLSchema#>]>
```

```
<rdf:RDF
```

```
  xmlns : rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns : xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns : uni="http://www.mydomain.org/uni-ns#"
  <rdf:Description rdf:about="949352">
    <uni : name>Grigoris Antoniou</uni : name>
    <uni : title>Professor </uni : title>
  </rdf : Description>
  <rdf:Description rdf:about="949318">
    <uni : name>David Billington</uni : name>
    <uni : title>Associate Professor </uni : title>
    <uni : age rdf:datatype="xsd;integer">27</uni : age>
  </rdf : Description>
```

```
<rdf:Description rdf:about="949111">
  <uni : name>Michael Maher</uni : name>
  <uni : title>Professor </uni : title>
</rdf : Description>
```

```
<rdf:Description rdf:about="CIT1111">
  <uni : CourseName>Discrete Mathematics</uni : courseName>
  <uni : isTaughtBy>David Billington</uni : isTaughtBy>
</rdf : Description>
```

```
<rdf:Description rdf:about="CIT1112">
  <uni : CourseName>Concrete Mathematics</uni : courseName>
  <uni : isTaughtBy>Grigoris Antonio</uni : isTaughtBy>
</rdf : Description>
```

```
<rdf:Description rdf:about="CIT2112">
  <uni : CourseName>Programming III</uni : courseName>
  <uni : isTaughtBy>Michael Macher</uni : isTaughtBy>
</rdf : Description>
```

```
<rdf:Description rdf:about="CIT3112">
  <uni : CourseName>Theory of Computation</uni : courseName>
  <uni : isTaughtBy>David Billington</uni : isTaughtBy>
</rdf : Description>
```

```
<rdf:Description rdf:about="CIT3116">
  <uni : CourseName>Knowledge representation</uni : courseName>
  <uni : isTaughtBy>Grigoriou Antoniou</uni : isTaughtBy>
</rdf : Description>
```

```
</rdf : RDF>
```

Note. This example is slightly misleading (for illustrative purposes) because we should replace all occurrences of course and staffID's such as 94952 and CIT3112 by references to the external namespaces, for example

```
<rdf : Description
  rdf : about=http://www.mydomain.org/uni-ns/#CIT31112>
```

The `rdf:resource` Attribute

- The preceding example was not satisfactory in one respect:
 - The relationship between courses and lecturers were not formally defined but existed implicitly through the use of the same name
 - To a machine, the use of the same name may just be a coincidence:
 - e.g., the David Billington who teaches CIT3112 may not be the same person as the person with ID 94318 who happens to be called David Billington
 - This problem can be avoided by using *rdf:resource* attribute as follows:

```
<rdf:Description rdf:about="CIT1111">  
    <uni : CourseName>Discrete Mathematics</uni : courseName>  
    <uni : isTaughtBy rdf:resource="949318"/>  
</rdf : Description>
```

```
<rdf:Description rdf:about="949318">  
    <uni : name>David Billington</uni : name>  
    <uni : title>Associate Professor </uni : title>  
</rdf : Description>
```

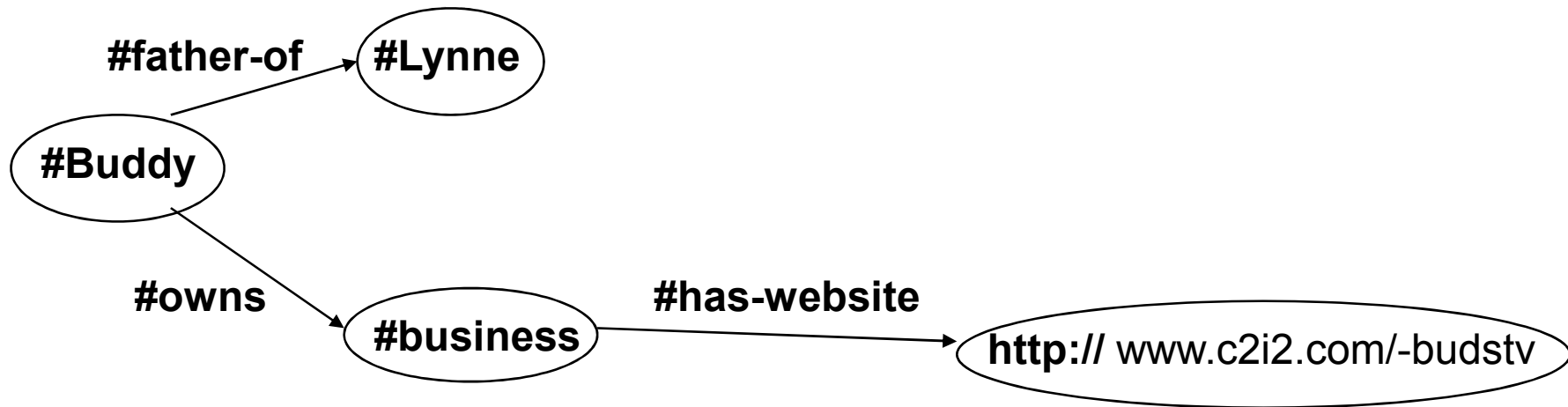

The rdf:type Element

- In our university example the descriptions fall into two categories: courses and lecturers. This fact is clear to human readers, but has not been formally declared anywhere, so it is not accessible to machines.
- In RDF it is possible to make such statements (connections to RDF Schema) using RDF : type element, e.g., as follows

```
<rdf:Description rdf:about="CIT1111">  
<rdf:type rdf:resource="&uni;course"/>  
  <uni : CourseName>Discrete Mathematics</uni : courseName>  
  <uni : isTaughtBy>David Billington</uni : isTaughtBy>  
</rdf : Description>
```

```
<rdf:Description rdf:about="949318">  
<rdf:type rdf:resource="&uni;lecturer"/>  
  <uni : name>David Billington</uni : name>  
  <uni : title>Associate Professor </uni : title>  
</rdf : Description>
```

Expressing contents by a graph of N3 notation



Other RDF features

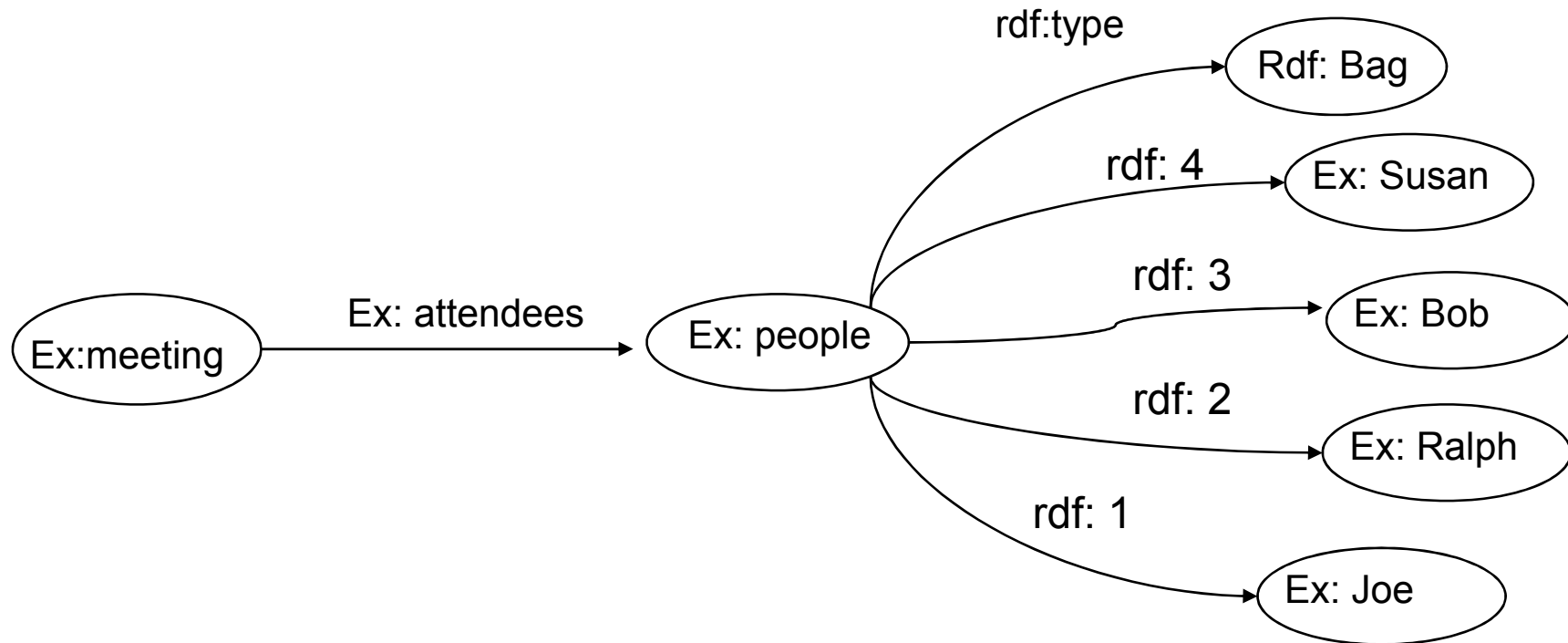
- The container model (Bag, Sequence, Alternate) allows groups of resources or values
 - Required to model sentences like “ The people at meeting were Joe, Bob, Susan, and Ralph
 - To model the objects in the sentence, a container, called bag is created (see next slide)

An RDF bag container example

```
<rdf:RDF
  xmlns:ex = 'http://www.example.org/sample#'
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>

  <rdf:Description rdf:about='ex:meeting*'>
    <ex:attendees>
      <rdf:Bag rdf:ID = "people">
        <rdf:li rdf: resource='ex:Joe'/>
        <rdf:li rdf: resource='ex:Bob'/>
        <rdf:li rdf: resource='ex:Susan'/>
        <rdf:li rdf: resource='ex:Ralph'/>
      </rdf: Bag>
    </ex: attendees>
  </rdf: Description>
</rdf: RDF>
```

Graph of an RDF bag



RDF containers

- Three types of RDF containers are available to group resources or literals:
- Bag
 - An `rdf: bag` element is used to denote an unordered collection, (duplicates are allowed)
- Sequence
 - An `rdf: seq` element is used to denote an ordered collection (a sequence of elements)
- Alternate
 - An `rdf: alt` element is used to denote a choice of multiple values or resources (e.g., a choice of image formats (JPEG, GIF, BMP))

Reification

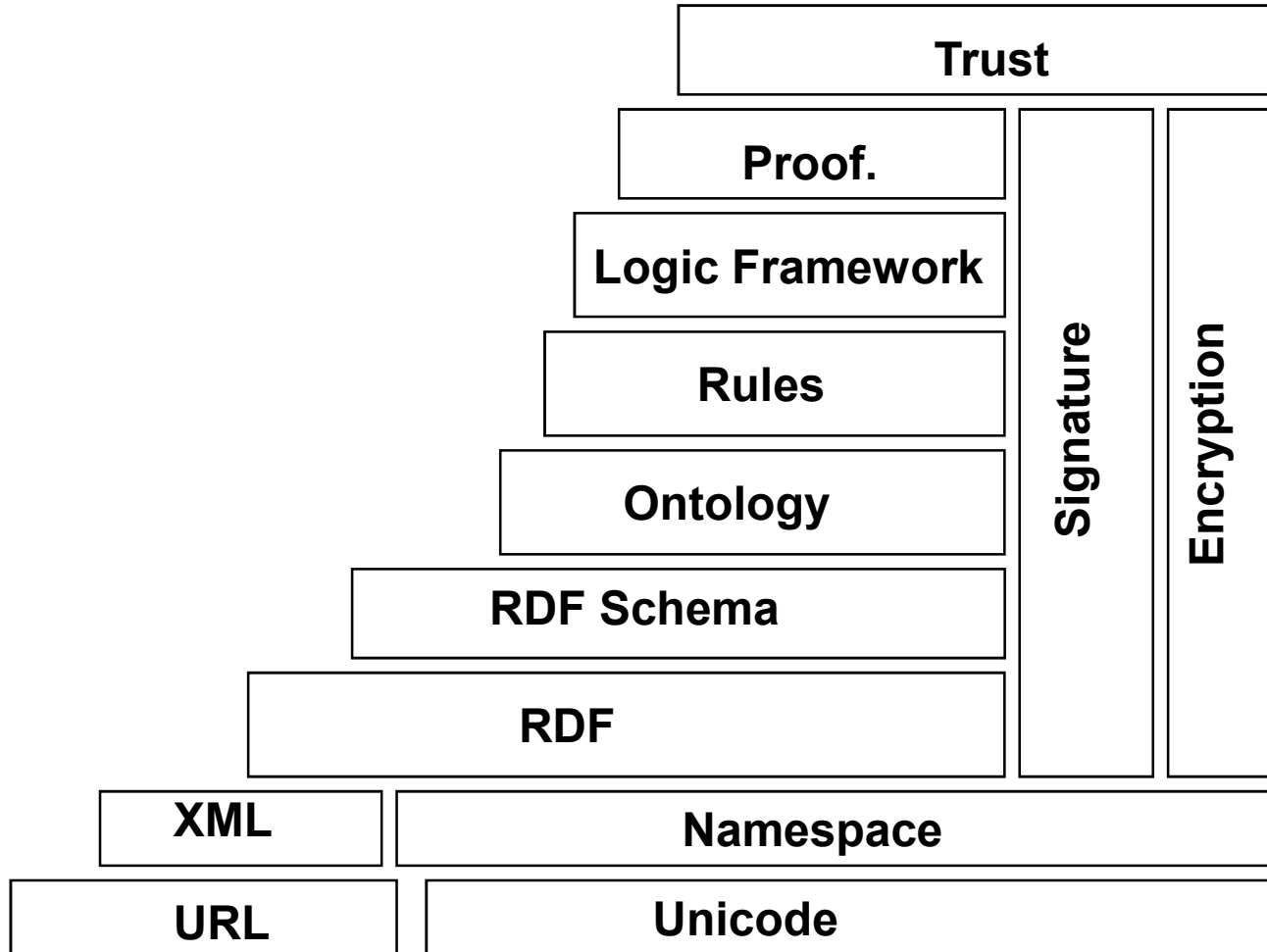
- Reification allows higher-level statements to capture knowledge about other statements
 - Reification mechanism turns a statement into a resource
 - Requires that we must be able to refer to a statement using an identifier, e.g., the description

```
<rdf : Description rdf : about = "949352">  
  <uni : name>Grigoris Antoniou</uni : name>  
</rdf : Description>
```

reifies as

```
<rdf : Statement rdf : about="StatementAbout949352">  
  <rdf : subject rdf : resource=="949352"/>  
  <rdf : predicate rdf : resource="&uni;name"/>  
  <rdf : object>Grigoris Antoniou</rdf:object>  
</rdf:Statement>
```

The Semantic Web Stack



RDF-Schema

- RDF is a universal language that lets user describe resources using their own vocabularies
- RDF does not make assumptions about any particular application domain, nor does it define the semantics of any domain. It is up to the user to do so in RDF Schema (RDFS)
- The data model expressed by RDF Schema is the same data model used by object-oriented programming languages like Java
 - A *class* is a group of things with common characteristics (properties)
 - In object oriented programming, a class is defined as a template or blueprint for an object composed of characteristics (also called data members) and behaviors (also called methods)
 - An *object* is an instance of a class
 - OO languages also allow classes to inherit characteristics (properties) and behaviors from a parent class (also called a super class)

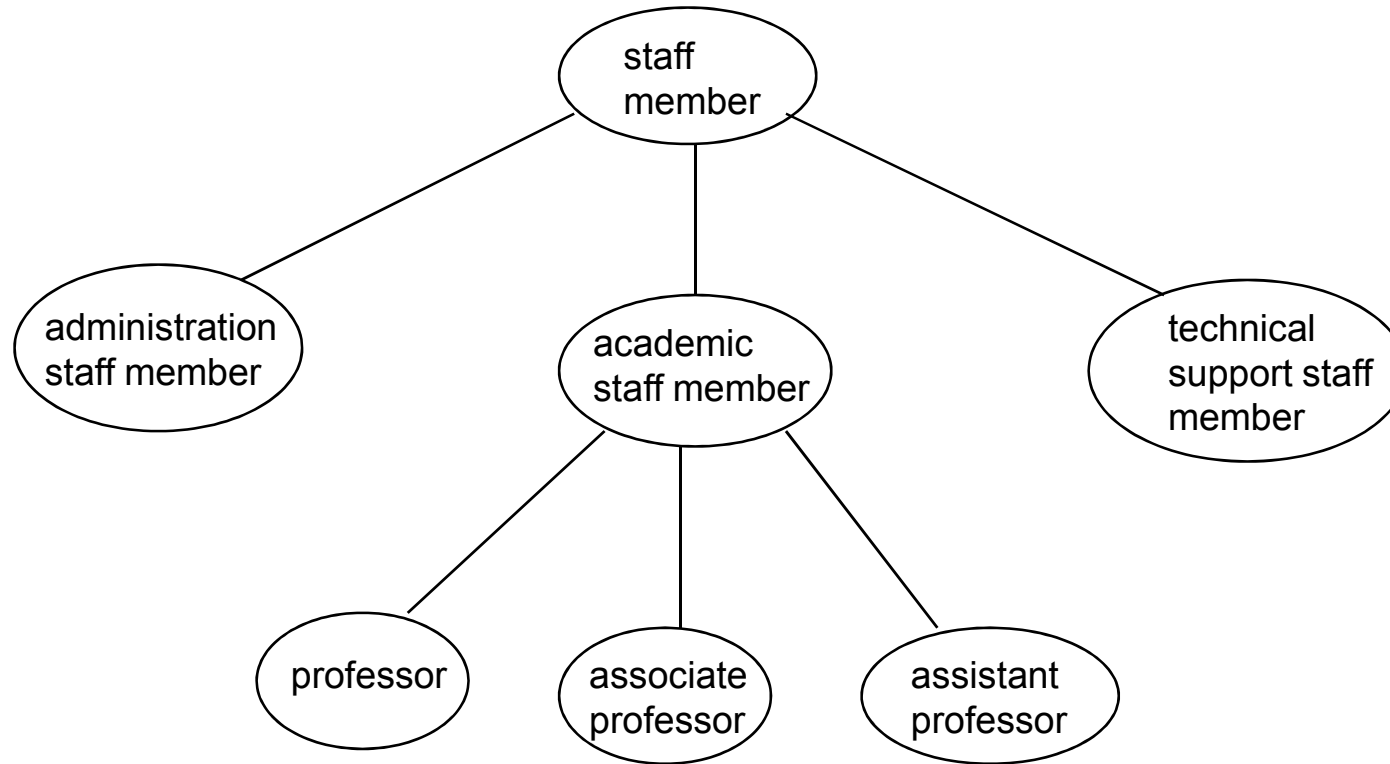
Classes and Properties

- How do we describe a particular domain; e.g., courses and lecturers in a university ?
 - First we have to specify “things” we want to talk about
 - We have already done so in RDF statements, but we restricted our talk on *individual objects* (resources)
 - Now, in the context of RDFS we talk about *classes* that define *types of individual objects*
- An important use of classes is to impose restrictions on what can be stated in an RDF document using the schema; e.g., we would like to disallow statements such as:
 - Discrete Mathematics is taught by Concrete Mathematics
 - Room MZH5760 is taught by David Billington

Class Hierarchies and Inheritance

- Once we have classes we would like to establish relationships between them
 - For example, assume that we have classes for
 - staff members
 - academic staff members
 - professors
 - associate professors
 - assistant professors
 - administrative staff members
 - technical support staff members
 - These classes are related to each other; e.g., every professor is an academic staff member, i.e.,
“professor” is *subclass* of “academic staff member”,
or equivalently,
“academic staff member” is a *superclass* of a “professor”

The subclass relationship defines a hierarchy of classes:



- In general, *A* is a subclass of *B* if every instance of *A* is also an instance of *B*.
 - However, there is no requirement in RDF Schema that the classes together form a strict hierarchy
 - So, a class may have multiple superclasses meaning that if a class *A* is a subclass of both *B1* and *B2*, then every instance of *A* is both an instance of *B1* and an instance of *B2*

- A hierarchical organization of classes has a very important practical significance. To illustrate this consider the *range restriction*:

Courses must be taught by academic staff members only

Assuming that Michael Maher were defined as a professor, then according to this restriction he is not allowed to teach courses. The reason is that there is no statement specifying that Michael Maher is also an academic staff member

We would like Michael Maher to *inherit* the ability to teach from the class of academic staff members. This is done in RDF Schema.

By doing so RDF Schema fixes the semantics of “is subclass of”, So, it is not up to an application to interpret “is a subclass of”; instead its intended meaning must be used by all RDF processing software.

- Classes, inheritance, and properties are, of course, known in other fields of computing, e.g., in object oriented programming
 - The differences are that in object oriented programming an object class defines the properties that apply to it. Also to add new properties to a class means to modify the class.
 - In RDFS, properties are defined globally, i.e., they are not encapsulated as attributes in class definitions. It is possible to define new properties that apply to an existing class without changing that class.
 - This is a powerful mechanism with far-reaching consequences: we may use classes defined by others and adapt them to our requirements through new properties

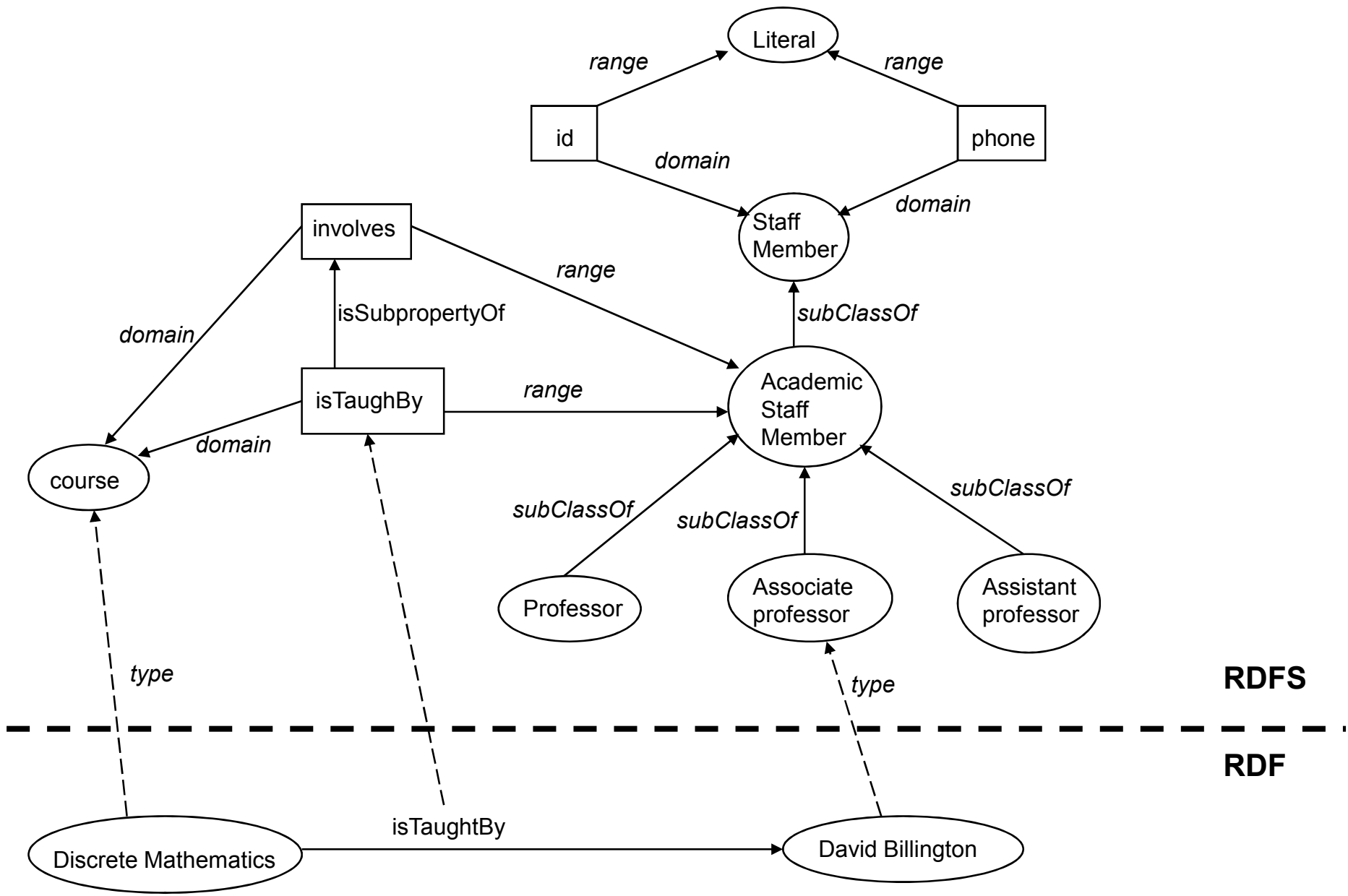
Property Hierarchies

- Hierarchical relationship can be defined not only for classes but also for properties:
 - Example, “is taught by “ is a *subproperty* of “involves”
So, if a course c is taught by an academic staff member a , then c also involves a
 - The converse is not necessary true, e.g., a may be the convener of the course, or a tutor who marks student home work but does not teach c
 - In general, P is a subproperty of Q if $Q(x,y)$ whenever $P(x,y)$

RDF versus RDFS Layers

- We illustrate the different layers involved in RDF and RDFS using a simple example:
 - Consider the RDF statement

Discrete Mathematics is taught by David Billington
 - The schema for this statement may contain classes, such as lecturers, academic staff members, staff members, and properties such as is taught by, involves, phone, employee id
 - Next figure illustrates the layers of RDF and RDF Schema (blocks are properties, ellipses above the dashed line are classes, and ellipses below the dashed line are instances)
 - The schema of the figure is itself written in a formal language, RDF Schema, that can express its ingredients: *subClass*, *Property*, *subProperty*, *Resource*, and so on.



RDFS

RDF

RDF Schema: The language

- RDF Schema provides *modeling primitives* for expressing the information of the previous figure
- One decision that must be made is what formal language to use?
 - RDF itself will be used: the modeling primitives of RDF Schema are defined using resources and properties
- **Note.** RDF allows one to express any statement about any resource, and that anything that has a URI can be a resource.
 - So, if we wish to say that the class “lecturer” is a subclass of “academic staff member”, we may
 1. Define resources *lecturer*, *academicStaffMember*, and *subClassOf*
 2. Define *subClassOf* to be a property
 3. Write the triple (*subClassOf*, *lecturer*, *academicStaffMember*)
 - **All these steps are within the capabilities of RDF.** So an RDFS document (that is an RDF schema) is just an RDF document, and we use the XML-based syntax of RDF.

RDF Schema: Core Classes

- The core classes are

rdfs : Resource, the class of resources

rdfs : Class, the class of classes

rdfs : Literal, the class of all literals (strings). At present, literals form the only “data type” of RDF/RDFS

rdf : Property. The class of all properties

rdf : Statement, the class of all reified statements

For example, a class *lecturer* can be defined as follows:

```
<rdfs : Class rdf : ID="lecturer">
```

```
...
```

```
</rdfs : Class>
```

RDF Schema: Core Properties for Defining Relationships

- The core properties for defining relationships are:

rdf : type, which relates a resource to its class. The resource is declared to be an instance of that class

rdfs : subClassOf, which relates a class to one of its superclasses, all instance of a class are instances of its superclass

Note. A class may be a subclass of more than one class, e.g., the class *femaleProfessors* may be a subclass of both *female* and *professor*

rdfs : subpropertyOf, which relates a property to one of its superproperties

- **Example:** stating that all lecturers are staff members

```
<rdfs : Class rdf : about="lecturer">  
  <rdfs : subClassOf rdf : resource="staffMember"/>  
</rdfs : Class>
```

Note that *rdfs:subClassOf* and *rdfs:subPropertyOf* are transitive by definition

Also *rdfs:Class* is a subclass of *rdfs:Resource* (every class is a resource), and *rdfs:Resource* is an instance of *rdfs:Class* (*rdfs:Resource* is the class of all resources, and so it is a class). For the same reason, every class is an instance of *rdfs:Class*

RDF Schema: Core Properties for Restricting Properties

- The core properties for restricting properties are:

rdfs:domain, which specifies the domain of a property *P*, i.e., the class of those resources that may appear as subject in triple with predicate *P*. If the domain is not specified, then any resource can be the subject

rdfs:range, which specifies the range of a property *P*, i.e., the class of those resources that may appear as values in a triple with predicate *P*

Example: stating that phone applies to staff members only and that its value is always a literal

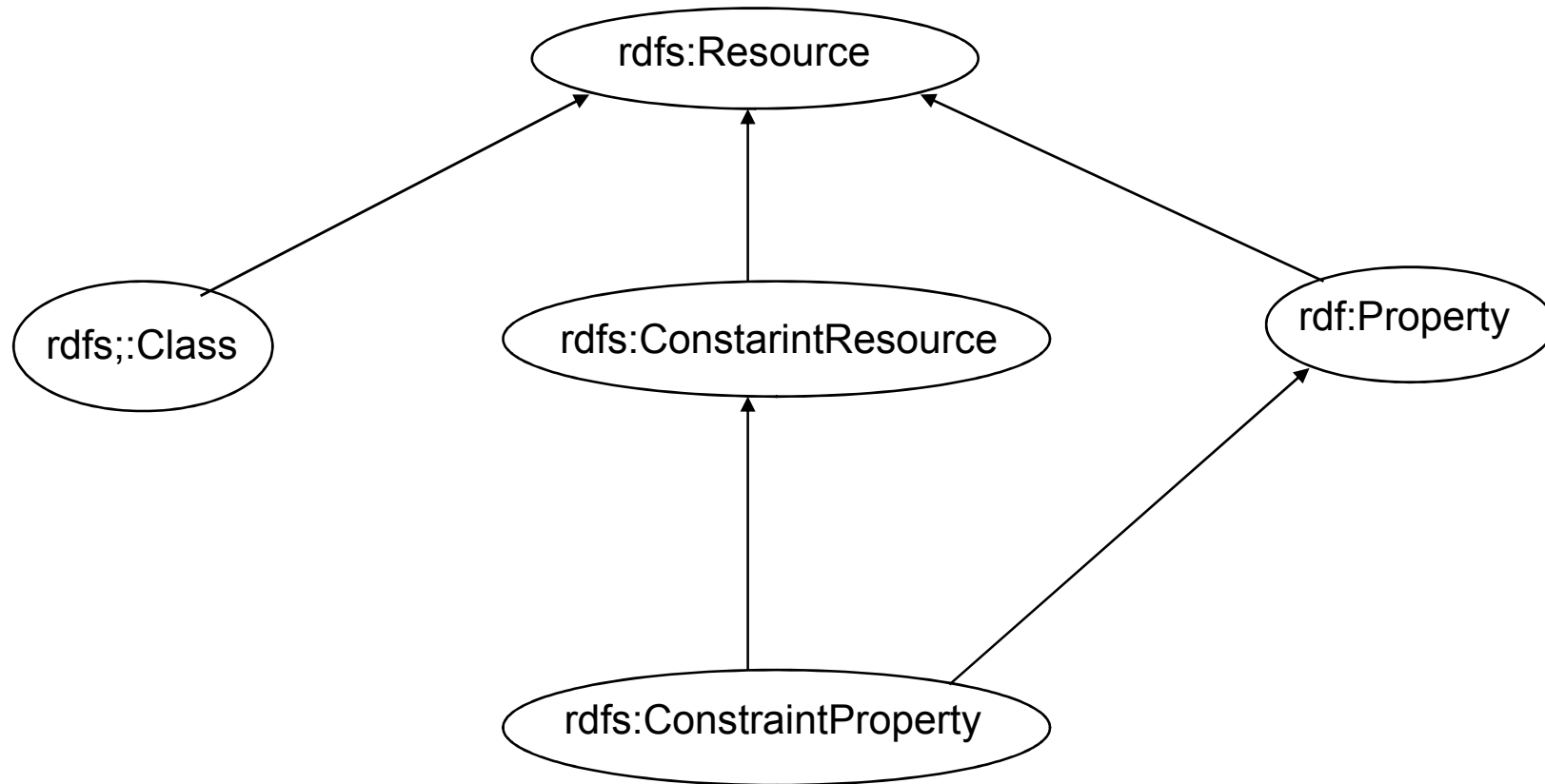
```
<rdfs:Property rdf:ID="phone">  
  <rdfs:domain rdf:resource="#staffMember"/>  
  <rdfs:range rdf:resource="&rdf;Literal"/>  
</rdf:Property>
```

- In RDF Schema there are also

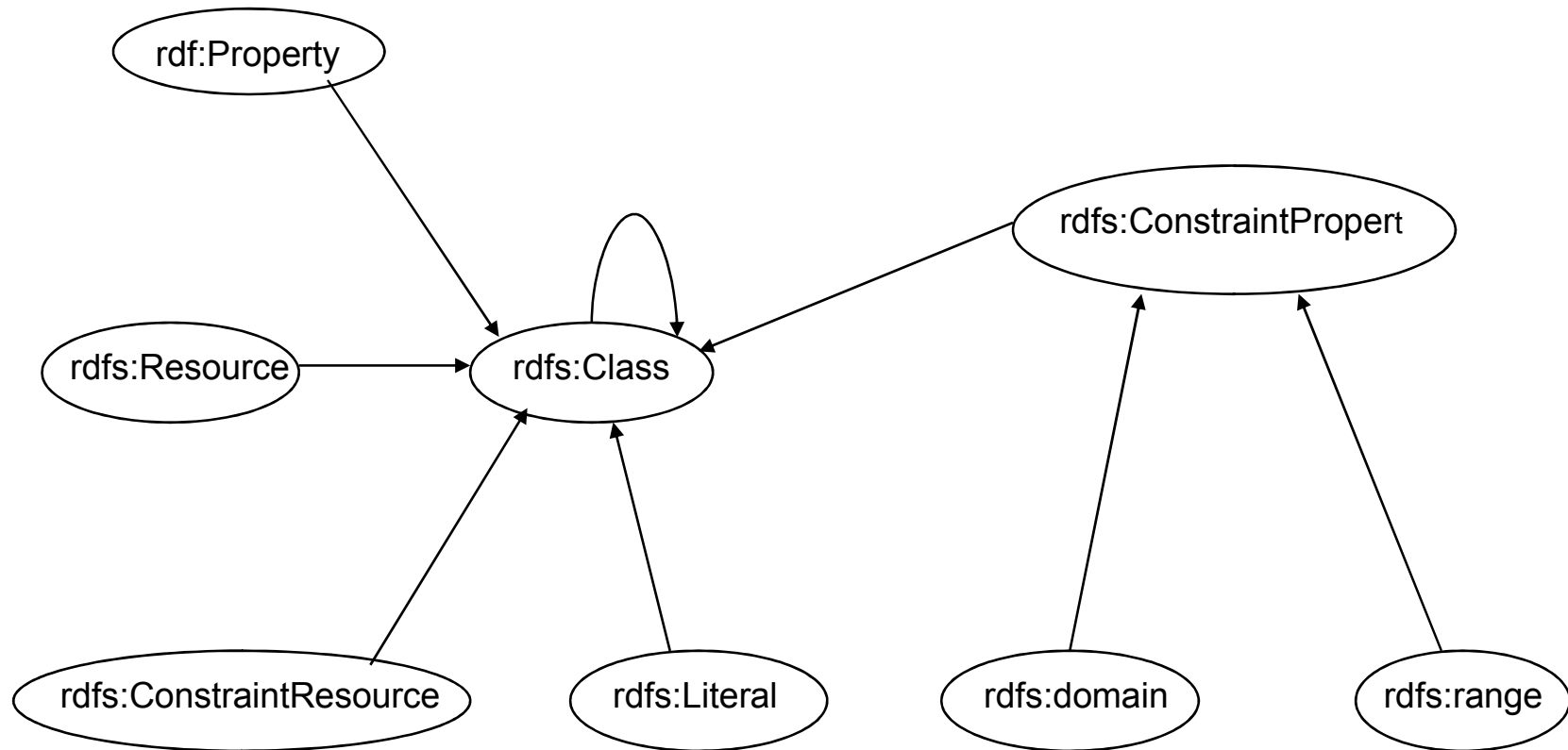
rdfs:ConstraintResource, the class of all constraints

rdfs:ConstraintProperty, which contains all properties that define constraints. It has only two instances, *rdfs:domain* and *rdfs:range*. It is defined to be a subclass of *rdfs:ConstraintResource* and *rdf:Property*

Subclass hierarchy of some modeling primitives of RDFS



Instance relationships of some modeling primitives of RDFS



RDFS example: A University

- We refer to the courses and lecturers example, and provide a conceptual model of the domain, that is, an ontology

```
<rdf:RDF
```

```
  xmlns:rdf="http.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http.w3.org/2000/01/rdf-schema#">
```

```
  <rdfs:Class rdf:ID="lecturer">
```

```
    <rdfs:comment>
```

```
      The class of lecturers
```

```
      All lecturers are academic staff members.
```

```
    </rdfs:comment>
```

```
    <rdfs:subClassOf rdf:resource="academicStaffMember"/>
```

```
  </rdfs:Class>
```

```
<rdfs:Class rdf:ID="academicStaffMember">
  <rdfs:comment>
    The class of academic staff members
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="staffMember"/>
</rdfs:Class>

<rdfs:Class rdf:ID="staffMember">
  <rdfs:comment>The class of staff members</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="course">
  <rdfs:comment>The class of courses</rdfs:comment>
</rdfs:Class>

<rdfs:Property rdf:ID="involves">
  <rdfs:comment>
    It relates only courses to lecturers
  </rdfs:comment>
  <rdfs:domain rdf:resource="#course"/>
  <rdfs:range rdf:resource="#lecturer"/>
</rdfs:Property>
```

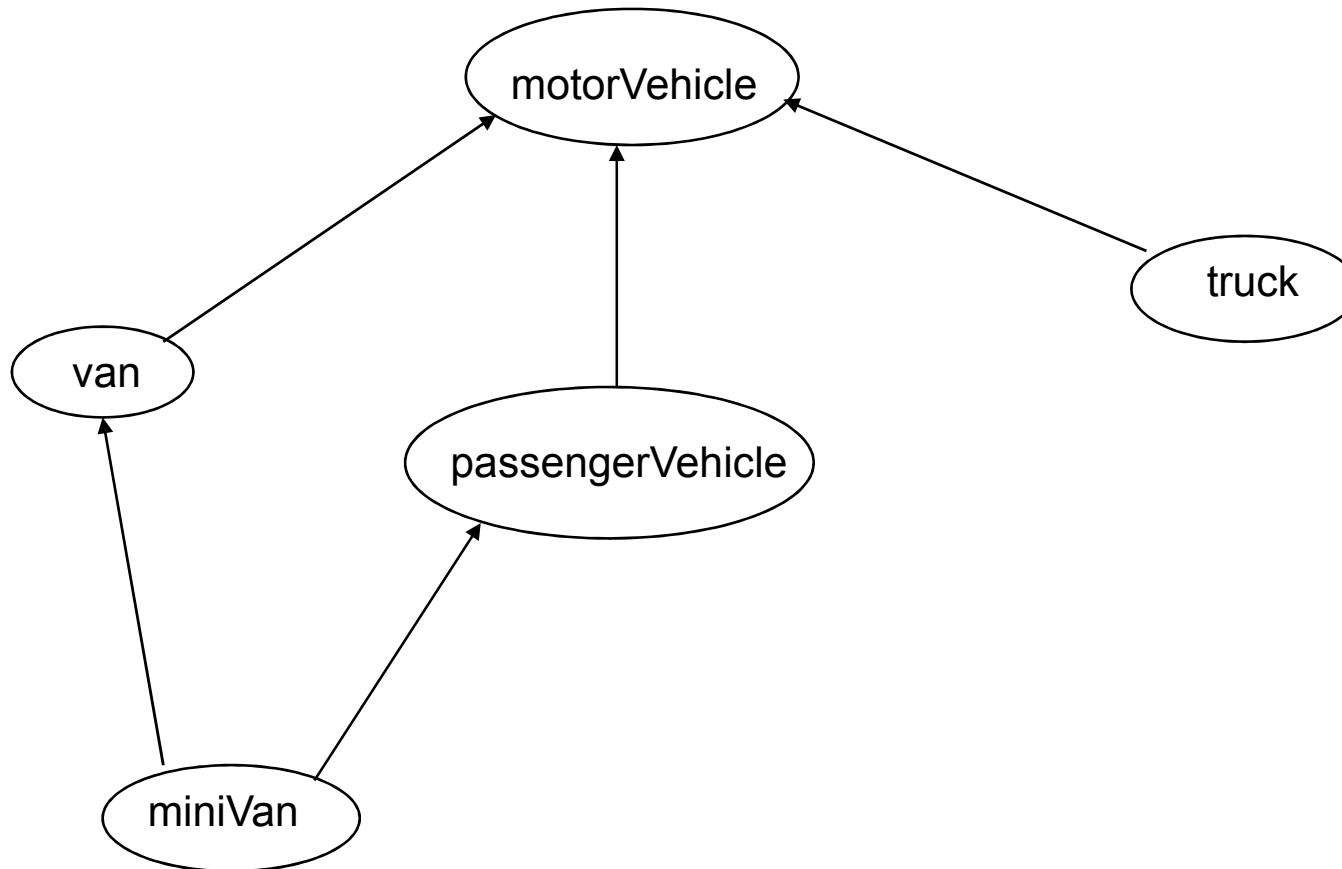
```
<rdfs:Property rdf:ID="isTaughtBy">
  <rdfs:comment>
    Inherits the domain ("course") and range ("lecturer") from its superproperty
    "involves"
  </rdfs:comment>
  <rdfs:subpropertyOf rdf:resource="#involves"/>
</rdfs:Property>
```

```
<rdfs:Property rdf:ID="phone">
  <rdfs:comment>
    It is a property of staff members and takes literals as values.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#staffMember"/>
  <rdfs:range rdf:resource="&rdf;Literal"/>
</rdfs:Propert>
```

```
</rdf:RDF>
```

Example: Motor Vehicles

The class relationships are the following:



```
<rdf:RDF
  xmlns:rdf="http.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http.w3.org/2000/01/rdf-schema#">

  <rdfs:Class rdf:ID="motorVehicle"/>

  <rdfs:Class rdf:ID="van">
    <rdfs:subClassOf rdf:resource="#motorVehicle"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="truck">
    <rdfs:subClassOf rdf:resource="#motorVehicle"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="passengerVehicle">
    <rdfs:subClassOf rdf:resource="#motorVehicle"/>
  </rdfs:Class>

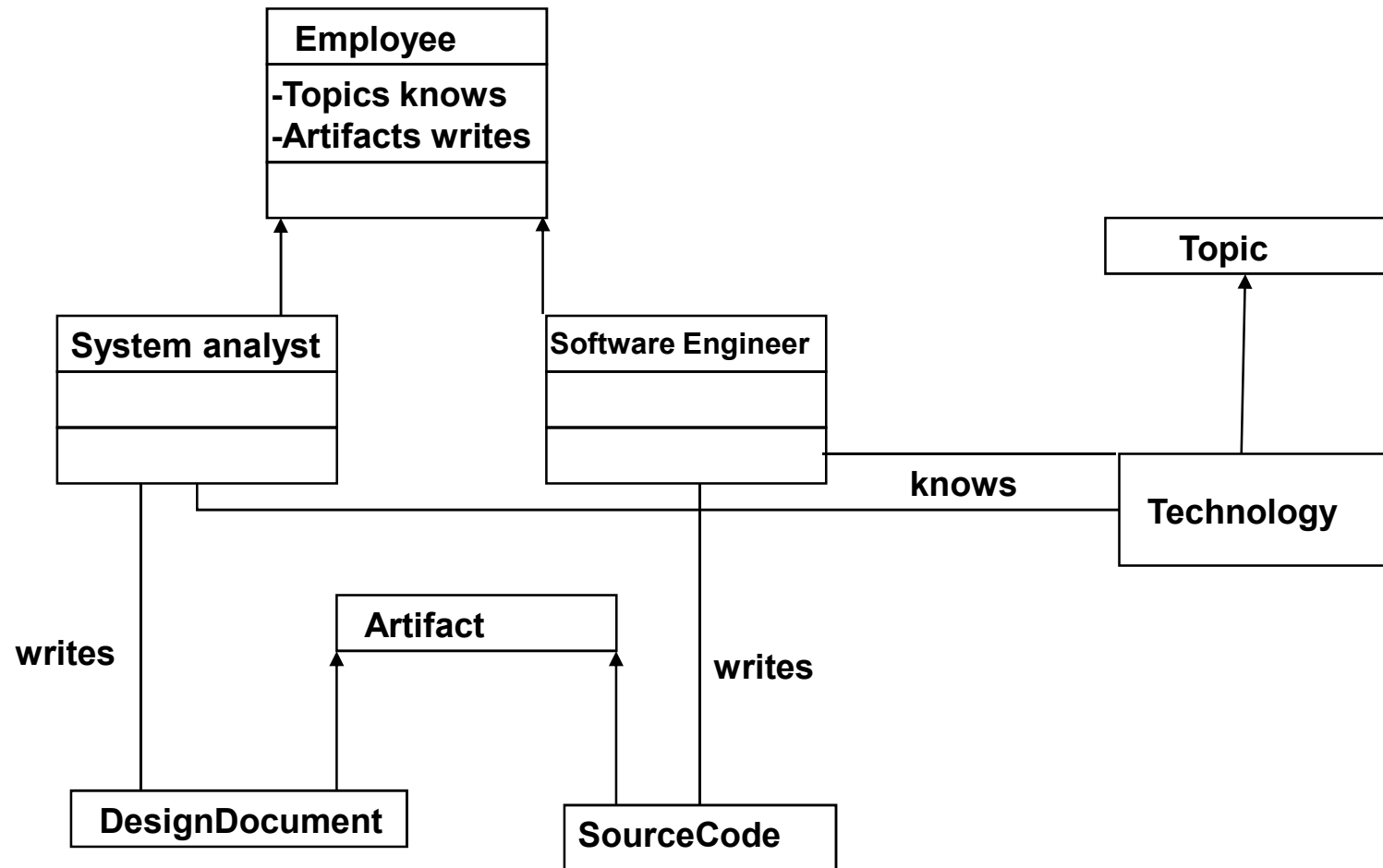
  <rdfs:Class rdf:ID="miniVan">
    <rdfs:subClassOf rdf:resource="#passengerVehicle"/>
    <rdfs:subClassOf rdf:resource="#van"/>
  </rdfs:Class>
</rdf:RDF>
```

RDFS versus UML (Unified Modeling Language)

- Standardized notation to model class hierarchies
- UML symbols denote the concepts of *class*, *inheritance*, and *association*
 - The rectangle, with three sections is the symbol for a class
 - The sections are: class name, the class attributes (middle section), and class behaviors or methods (bottom section)
 - The RDF Schema only uses the first two parts of a class, since it is used for data modeling and not programming behaviors
- An arrow from the subclass to the superclass denotes inheritance (a subclass inherits the characteristics of a superclass), also called “isa” (is a) in software engineering

UML class diagram of employee expertise

- Two types of employees and their associations to the artifacts they write and the topics they know



Chapter 6: Understanding XML – Related Technologies

Addressing and querying XML Documents

- In relational databases, parts of database can selected and retrieved using query languages such as SQL.
- The same is true for XML documents, for which there exist a number of proposals :

- XPath
 - standard addressing mechanism for XML nodes
- XSL
 - transforming and translating XML -documents
- XSLT
 - transforming and translating XML -documents
- XSLFO
 - transforming and translating XML -documents
- XQuery
 - Querying mechanism for XML data stores (“The SQL for XML”)
- XLink
 - general, all-purpose linking specification

XML –Related Technologies, continues ...

- XPointer
 - addressing nodes, ranges, and points in local and remote XML documents
- XInclude
 - used to include several external documents into a large document
- XML Base
 - Mechanism for easily resolving relative URIs
- XHTML
 - A valid and well formed version of HTML
- XForms
 - An XML-based form processing mechanism
- SVG
 - XML-based rich-content graphic rendering
- RQL
 - Query language for RDF

XPath

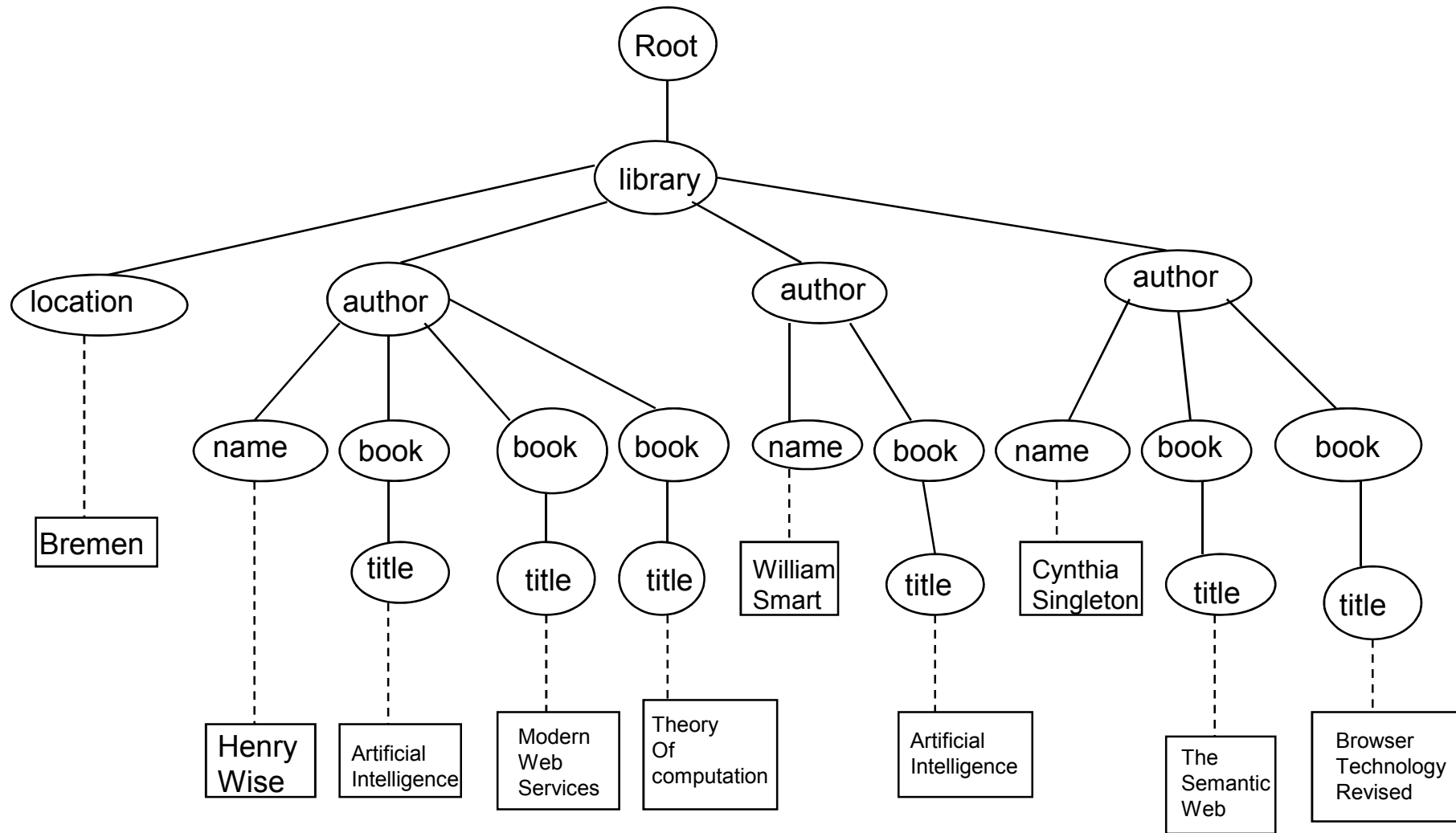
- The central concept of XML query languages is *path expression* that specifies how a node, or a set of nodes, in the tree representation of the XML document can be reached
- We consider path expressions in the form of XPath because they can be used for purposes other than querying, namely transforming XML documents
- XML Path Language - an expression language for specifically addressing parts of an XML document
- Provides key semantics, syntax, and functionality for a variety of standards, such as XSLT, XPointer, and XQuery
- By using XPath expressions with certain software frameworks and APIs, it is possible to reference and find the values of individual components of an XML document

XPath

- XPath operates on the tree data model of XML and has a non-XML syntax
- Path expression can be
 - Absolute (starting at the root of the tree); syntactically they begin with the symbol /, which refers to the root of the document, situated one level above the root element of the document
 - Relative to context node
- Consider the following XML-document

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE library PUBLIC "library.dtd">
<library location="Bremen">
  <author name="Henry Wise">
    <book title="Artificial Intelligence"/>
    <book title="Modern Web Services"/>
    <book title="Theory of Computation"/>
  </author>
  <author name="William Smart">
    <book title="Artificial Intelligence"/>
  </author>
  <author name="Cynthia Singleton">
    <book title="The Semantic Web"/>
    <book title="Browser Technology Revised"/>
  </author>
</library>
```

Tree representation of the library document



Examples of XPath expressions on the library document

- Address all author elements

```
/library/author
```

An alternative solution for the previous example:

```
//author
```

Here `//` says that we should consider all elements in the document and check whether they are type *author*

Note, because of the specific structure of the library document these expressions lead to the same result, however they may lead to different result in general

More examples of XPath expressions on the library document

- Address the *location* attribute nodes within *library* element nodes

```
//library/@location
```

The symbol @ is used to denote attribute nodes

- Address all *title* attribute within book elements anywhere in the document, which have the value “*Artificial Intelligence*”

```
//book/@title="Artificial Intelligence"
```

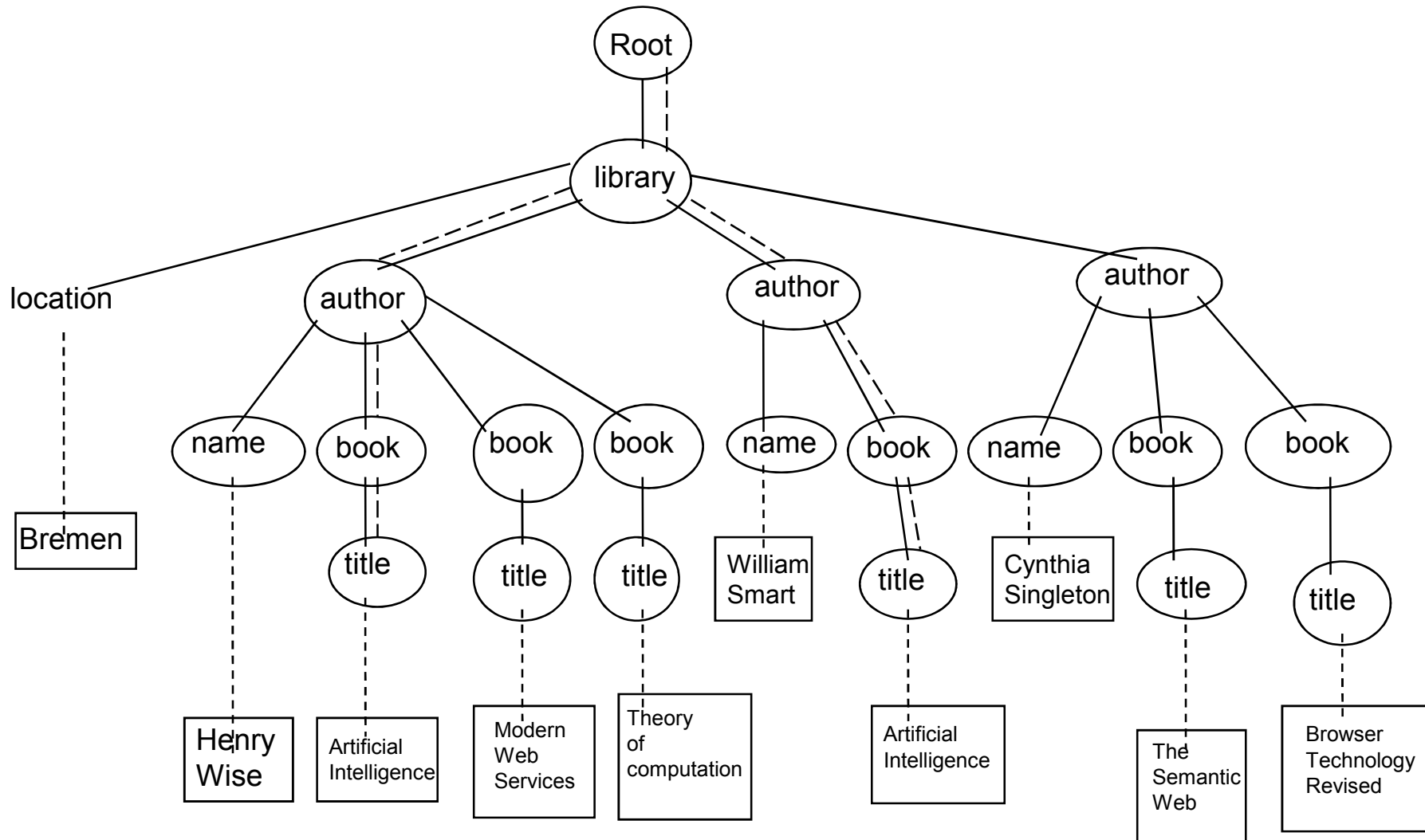
- Address all books with title “*Artificial Intelligence*”

```
//book [ @title="Artificial Intelligence"]
```

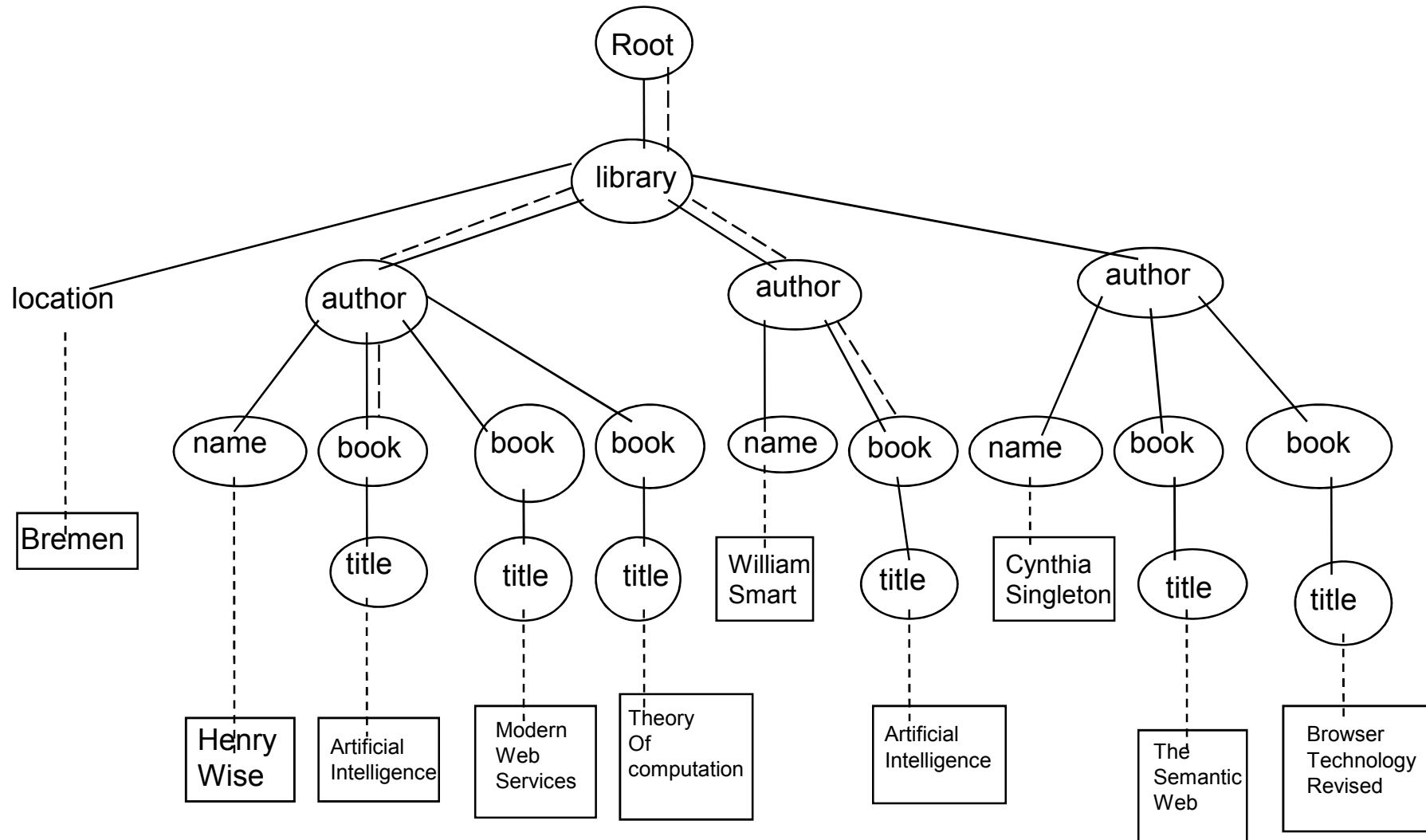
The test within square brackets is a filter expression. It restricts the set of addressed nodes

Note the difference between this and the one in previous query. Here we address *book* elements the title of which satisfies a certain condition. In previous query we collect title attribute nodes of book elements. The comparison of these queries is illustrated in the following two figures.

Query: //book/@title="Artificial Intelligence"



Query: //book [@title="Artificial Intelligence"]



Examples of XPath expressions on the library document

- Address the last *book* element within the first *author* element node in the document

```
//author[1] / book [last () ]
```

- Address all book element nodes without a title attribute

```
//book [not @title]
```

More examples of XPath expressions and return values

```
<Task>
  <TaskItem id = "123"
    value = " Status Report"/>
  <TaskItem id = "124"
    value = " Writing Code"/>
  <TaskItem value ="IdleChat"/>
  <Meeting id = "125"
    value= "Daily Briefings"/>
</Task>
```

XPath expression	Meaning	Return values
<code>//TaskItem[@id]</code>	"Give me all TaskItem elements that have ID attributes"	<pre><TaskItem id = "123" value = " Status Report"/> <TaskItem id = "124" value = " Writing Code"/></pre>

Examples of XPath expressions and return values

```
<Task>
  <TaskItem id = "123"
    value = " Status Report"/>
  <TaskItem id = "124"
    value = " Writing Code"/>
  <TaskItem value ="IdleChat"/>
  <Meeting id = "125"
    value= "Daily Briefings"/>
</Task>
```

XPath expression	Meaning	Return values
<code>//[@id]</code>	"Give me all ID attributes"	Id = "123" Id = "124" Id = "125"

Examples of XPath expressions and return values

```
<Task>
  <TaskItem id = "123"
    value = " Status Report"/>
  <TaskItem id = "124"
    value = " Writing Code"/>
  <TaskItem value ="IdleChat"/>
  <Meeting id = "125"
    value= "Daily Briefings"/>
</Task>
```

XPath expression	Meaning	Return values
/Task/Meeting	“Select all elements named ‘Meeting’ that are children of the root element ‘Task’ ”	<Meeting id = “125” value= “Daily Briefings”/>

The structure of XPath expression

- A path expression consists of a series of steps, represented by slashes
- A step consists of an axis specifier, a node test, and an optional predicate
 - An *axis specifier* determines the tree relationship between the nodes to be addressed and the context node. Examples are parent, ancestor, child (the default), sibling, attribute node. // is such an axis specifier; it denotes descendant or self.
 - A *node test* specifies which nodes to be addressed. The most common node tests are element names
 - *Predicates (or filter expressions)* are optional and are used to refine the set of addressed nodes.

The role of XPath in other standards

- With XSLT one can define a template in advance using XPath expressions that allow to specify how to style a document
- XQuery is a superset of XPath and uses XPath expressions to query XML native databases and multiple XML files
- XPointer uses XPath expressions to “point” specific nodes in XML documents
- XML Signature, XML Encryption, and many other standards can use XPath expressions to reference certain areas of an XML document
- In a Semantic Web ontology, groups of XPath expressions can be used to specify how to find data and the relationship between data.

RQL

- A query language for RDF
- RQL is needed because XML query languages locate at a lower level of abstraction than RDF
- Querying RDF documents with an XML-based language would lead to complications.
 - To illustrate this let us consider different syntactical variations of writing an RDF description:

```
<rdf:Description rdf:about="949318">  
  <rdf:type rdf:resource="&uni;lecturer"/>  
  <uni:name>David Billington</uni:name>  
  <uni:title>Associate Professor</uni:title>  
</rdf:Description>
```

Now assume that we wish to retrieve the titles of all lecturers. An appropriate XPath query is

```
/rdf:Description [rdf:type=http://www.mydomain.org/uni-ns#lecturer]/uni:title>
```

We could have written the same description as follows:

```
<uni:lecturer rdf:about="949318">  
  <uni:name>David Billington</uni:name>  
  <uni:title>Associate Professor</uni:title>  
</rdf:Lecturer>
```

Now the previous XPath query does not work; instead we have to write

```
//uni:lecturer/uni:title
```

The third possible representation of the same description is:

```
<uni:lecturer rdf:about="949318"  
  uni:name="David Billington"  
  uni:title="Associate Professor"/>
```

For this variation XPath query is

```
//uni:lecturer/@uni:title
```

- Since each description of an individual lecturer may have any of these equivalent forms, we have to write different XPath queries
- A better way is to write queries at the level of RDF
- An appropriate query language must understand RDF, i.e., it must understand:
 - The syntax of RDF
 - The data model of RDF,
 - The semantics of RDF vocabulary, and
 - The semantics of RDF Schema
- To illustrate this consider the information:

```
<uni:lecturer rdf:about="94352">
  <uni:name>Grigiris Antoniou</uni:name>
</rdf:Lecturer>

<uni:professor rdf:about="94318">
  <uni:name>David Billington</uni:name>
</rdf:professor>

<rdfs:Class rdf:about="&uni;professor">
  <rdfs:subClassOf rdf:resource="&uni;lecturer"/>
</rdfs:Class>
```

Now, the query for the names of all lecturers should return both Grigiris Antoniou and David Billington

RQL: Basic Queries

- The query *Class* retrieves all classes, and the query *Property* retrieves all properties
- To retrieve the instances of a class, e.g., *course*, we write:

course

- This query will return all instances of the subclass of *course*
- *If we do not wish to retrieve inherited instances then we have to write:*

^course

- The resources and values of triples with a specific property, e.g., *involves*, are retrieved using simply the query

involves

- The result includes all subproperties of *involves*, e.g., it retrieves also inherited triples from property *isTaughtBy*
- If we do not want these additional results then we have to write

^involves

RQL: Using select-from-where

- As in SQL:

select specifies the number and order of retrieved data
from is used to navigate through the data model
where imposes constraints on possible solutions

For example, to retrieve all phone numbers of staff members, we can write

```
select X, Y  
from {X}phone{Y}
```

Here X and Y are variables, and {X}phone{Y} represents a resource-property-value triple.

- To retrieve all lecturers and their phone numbers, we can write

```
select X, Y  
from lecturer{X} .phone{Y}
```

- Here *lecturer{X}* collects all instances of the class *lecturer* and binds the result to the variable *X*.
- The second part collects all triples with predicate *phone*. But there is an *implicit join* here, in that we restrict the second query only to those triples, the resource of which is in the variable *X*; in this example we restrict the domain of *phone* to lecturers.
- A dot *.* denotes the implicit join.

- We demonstrate an *explicit join* by a query that retrieves the name of all courses taught by the lecturer with ID 949352

```
select N  
from course{X} .isTaughtBy{y}, {C}name{N}  
where Y="949352" and X=C
```

- Apart from = there exist other *comparison operators*

For example, $X < Y$ means “X is lower than Y”

- In case X and Y are strings, X comes before Y in the lexicographic order
- If X and Y are classes, then X is a subclass of Y

RQL: Querying the schema

- RQL allows to retrieve schema information
- Schema variables have a name with prefix \$ (for classes) or @ (for properties)
 - For example

```
select X, $X, Y, $Y  
from {X:$X}phone{Y:$Y}
```

retrieves all resources and values of triples with property *phone*, or any of its subproperties, and their classes.

Note that these classes may not coincide with the defined domain and a range of *phone*, because they may be subclasses of the domain or range

For example, given

```
phone("949352", "5041")
type("949352", lecturer)
subclass(lecturer, staffMember)
domain(phone, staffMember)
range(phone, literal)
```

We get

```
("949352", lecturer, "5041", literal)
```

although *lecturer* is not the domain of *phone*

- The domain and range of a property can be retrieved as follows:

```
select domain (@P), range (@P)
from @P
where @P=phone
```

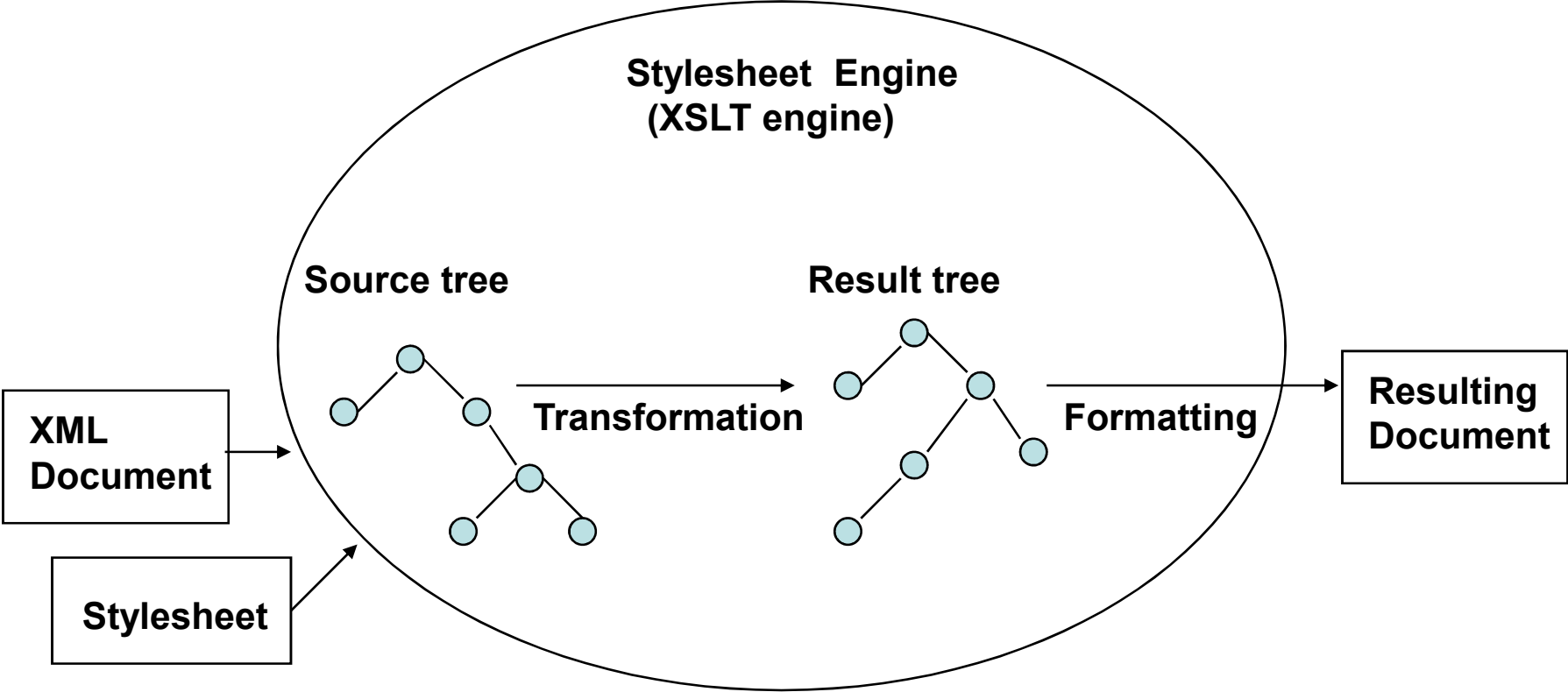
The Style Sheet Family: XSL, XSLT and XSLFO

- Style sheets allow to specify how an XML document can be transformed into new document, and how that XML document could be presented in different media formats
- A style sheet processor takes an XML document and a style sheet and produces a result
- XSL consists of two parts:
 - It provides a mechanism for transforming XML documents into new XML documents (XSLT), and
 - It provides a vocabulary for formatting objects (XSLFO)
- XSLT is a markup language that uses template rules to specify how a style sheet processor transforms a document
- XSLFO is a pagination markup language

Model – View - Controller (MVC) paradigm

- Idea of MVC: separating content (the XML data) from the presentation (the style sheet)
- The act of separating the data (the model), how the data is displayed (the view), and the framework used between them (the controller) provides maximum reuse of resources
- Eliminates the maintenance of keeping track of multiple presentation format for the same data
- Because browsers such as Microsoft Internet Explorer have style sheet processors embedded in them, presentation can dynamically be added to XML data at download time

Styling a document



Styling a document, continues ...

1. A style sheet engine takes an original XML document, loads it into DOM source tree, and transforms that document with the instructions given in the style sheet
2. The result is formatted, and the resulting document is returned
 - Although the original document must be well-formed document the resulting document may be any format
 - Many times the resulting document may be postprocessed
 - With XSLFO styling, a post processor is usually used to transform the result document into a different format, e.g., PDF or RTF

Example: using style sheets to add presentation to content

- Consider the XML-element

```
<author>  
  <name>Grigoris Antoniou</name>  
  <affiliation>University of Bremen</affiliation>  
  <email>ga@tzi.de</email>  
</author>
```

The output might look like the following, if a style sheet is used:

Grigoris Antoniou
University of Bremen
ga@tzi.de

Or it might appear as follows, if different style sheet is used:

Grigoris Antoniou
University of Bremen
ga@tzi.de

XSLT document that will be applied to the author element

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">

  <xsl : template match="author">
    <html>
      <head><title>An author</title></head>
      <body bgcolor="white">
        <b><xsl:value-of select="name"/></b><br>
        <xsl : value-of select="affiliation"/><br>
        <i><xsl : value-of select="email"/></i>
      </body>
    </html>
  </xsl : template>
</xsl : stylesheet>
```

The output (HTML document) of the style sheet when applied to the author element

```
<html>
  <head><title>An author</title></head>
  <body bgcolor="white">
    <b>Grigoris Antoniou</b><br>
    University of Bremen<br>
    <i>ga@zi.de</i>
  </body>
</html>
```

Some observations concerning the previous example

- The XSLT document defines a *template*; in this case an HTML document, with some placeholders for content to be inserted

```
<html>
  <head><title>An author</title></head>
  <body bgcolor="white">
    <b> . . . </b><br>
    . . . <br>
    <i> . . . </i>
  </body>
</html>
```

- In the XSLT document, *xsl : value-of* retrieves the value of an element and copies it into the output document, i.e., it places some content into the template

- Now suppose we had the following XML document with details of several authors

```
<authors>
  <author>
    <name>Grigoris Antoniou</name>
    <affiliation>University of Bremen</affiliation>
    <email>ga@tzi.de</email>
  </author>
  <author>
    <name>David Billington</name>
    <affiliation>Griffith University</affiliation>
    <email>david@gu.edu.net</email>
  </author>
</authors>
```

We define the following XSLT document for it:

```

<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">
  <xsl : template match="/">
    <html>
      <head><title>Authors</title></head>
      <body bgcolor="white">
        <xsl : apply-template select "authors"/>
        <!-- Apply templates for AUTHORS children -->
      </body>
    </html>
  </xsl : template>
  <xsl : template match="authors">
    <xsl : apply-templates select="author"/>
  </xsl : template>

  <xsl : template match="author">
    <h2><xsl : value-of select="name"/></h2>
    Affiliation: <xsl : value-of select="affiliation"/><br>
    Email: <xsl : value-of select="email"/>
  <p>
  </xsl : template>
</xsl : stylesheet>

```

- The output produced is

```
<html>
  <head><title>Authors</title></head>
  <body bgcolor="white">
    <h2>Grigoris Antoniou</h2>
    Affiliation: University of Bremen<br>
    Email: ga@zi.de
    <p>
    <h2>David Billington</h2>
    Affiliation: Griffith University <br>
    Email: david@gu.edu.net
    <p>
  </body>
</html>
```

Some observations concerning the previous example

- The `xsl : apply-templates` element causes all children of the context node to be matched against the selected path expression
 - For example, if the current template applies to `/` (i.e., the current context node is the root) then the element `xsl : apply-templates` applies to the root element, in this case, the *authors element* (*/ is located above the root element*)
 - If the current context node is the *author* element, then the element `xsl : apply-templates select="author"` causes the template for the *author* elements to be applied to all *author* children of the *authors* element
 - It is good practice to define a template for each element type in the document. Even if no specific processing is applied to certain elements

An other example: using style sheets to add presentation to content

A simple xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="simple.xsl" type="text/XSL"?>
<project name="Trumantruck.com">
  <description>Rebuilding a 1967 Chevy Pickup Truck</description>
  <schedule>
    <workday>
      <date>20000205</date>
      <description>Taking Truck Body Apart</description>
    </workday>
    <workday>
      <date>20000225</date>
      <description>Sandblasting, Dismantling Cab</description>
    </workday>
    <workday>
      <date>20000311</date>
      <description>Sanding, Priming Hood and Fender</description>
    </workday>
  </schedule>
</project>
```

Example: using style sheets to add presentation to content, continues...

- To create an HTML page with the information from the previous XML file, a style sheet must be written (next slide)
- The style sheet creates an HTML file with the workdays listed in an HTML table
- All pattern matching is done with XPath expressions
- The `<xsl:value-of>` element returns the value of items selected from an XPath expression, and each template is called by the XSLT processor if the current node matches the XPath expression in the match attribute

XSLT document of the example

```
<xsl:stylesheet xmlns:xsl=http://www.w3.org/TR/WD-xsl>
  <xsl:template match="/">
    <html>
      <TITLE> Schedule For
      <xsl:value-of select="/project/@name"/>
-   <xsl: value-of select="/project/description"/>
      </TITLE>
      <CENTER>
        <TABLE border="1">
          <TR>
            <TD><B>Date</B></TD>
            <TD><B>Description</B></TD>
          </TR>
          <xsl:apply-templates/>
        </TABLE>
      </CENTRE>
    </html>
  </xsl:template>
```

```

<xsl: template match="project">
  <H1> Project :
  <xsl: value-of select="@name"/>
</H1>
  <HR/>
  <xsl : apply-template/>
<xsl : template match = "schedule">
  <H2> Work Schedule</H2>

  <xsl:apply-templates/>
</xsl : template>
<xsl : template match = "workday">
  <TR>
    <TD>
      <xsl : value -of select ="date"/>
    </TD>
    <TD>
      <xsl : value-of select=" description"/>
    </TD>
  </TR>
</xsl : template>
</xsl : stylesheet>

```

The final layout of the document (shown by a browser)

Project:
Trumantruck.com

Work Schedule

Date	Description
2000025	Taking Truck Body Apart
2000225	Sandblasting Dismantling Cab
2000311	Sanding, Priming Hood and Fender

The needs of style sheets

- In an environment where interoperability is crucial, and where data is stored in different formats for different enterprises, styling is used to translate one enterprise format to another enterprise format
- In a scenario where we must support different user interfaces for many devices, style sheets are used to add presentation to content
- A wireless client, a Web client, a Java application client, a .Net application client, or any application can have different style sheets to present a customized view

XQuery

- Designed for processing XML data
- Intended to make querying XML-based data sources as easy as querying databases
- Human-readable query syntax
- Extensions of XPath (with few exceptions, every XPath expression is also an XQuery expression)
 - However XQuery provides human readable language that makes it easy to query XML-sources and combine that with programming language logic

Example: XQuery expression

```
Let $project := document("trumanproject.xml")/project
```

```
Let $day := $project/schedule/workday
```

```
Return $day sortby (description)
```

Example XML file

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?xml-stylesheet href="simple.xsl" type="text/XSL"?>
```

```
<project name="Trumantruck.com">
```

```
  <description>Rebuilding a 1967 Chevy Pickup Truck</description>
```

```
  <schedule>
```

```
    <workday>
```

```
      <date>20000205</date>
```

```
      <description>Taking truck Body Apart</description>
```

```
    </workday>
```

```
    <workday>
```

```
      <date>20000225</date>
```

```
      <description>Sandblasting, Dismantling Cab</description>
```

```
    </workday>
```

```
    <workday>
```

```
      <date>20000311</date>
```

```
      <description>Sanding, Priming Hood and Fender</description>
```

```
    </workday>
```

```
  </schedule>
```

```
</project>
```


The result of the XQuery expression:

```
<workday>
  <date>20000225></date>
  <description>Sandblasting, Dismantling Cab</description>
</workday>
<workday>
  <date>20000311></date>
  <description>Sanding, Priming Hood and Fender</description>
</workday>
<workday>
  <date>20000205></date>
  <description>Taking truck Body Apart</description>
</workday>
```

XHTML

- XHTML- Extensible Hypertext Markup Language is the reformulation of HTML into XML
- Was created for the purpose of enhancing current Web to provide more structure for machine processing
- Documents formatted by HTML are not intended for machine processing
- Because XHTML is XML, it provides structure and extensibility by allowing the inclusion of other XML-based languages with namespaces

Example: making the transition from HTML into XHTML

```
<HTML>
  <HEAD>
    <TITLE>Morning to-do list</TITLE>
  </HEAD>
  <BODY>
    <LI>Wake up
    <LI>Make bed
    <LI>Drink coffee
    <LI>Go to work
  </BODY>
</HTML>
```

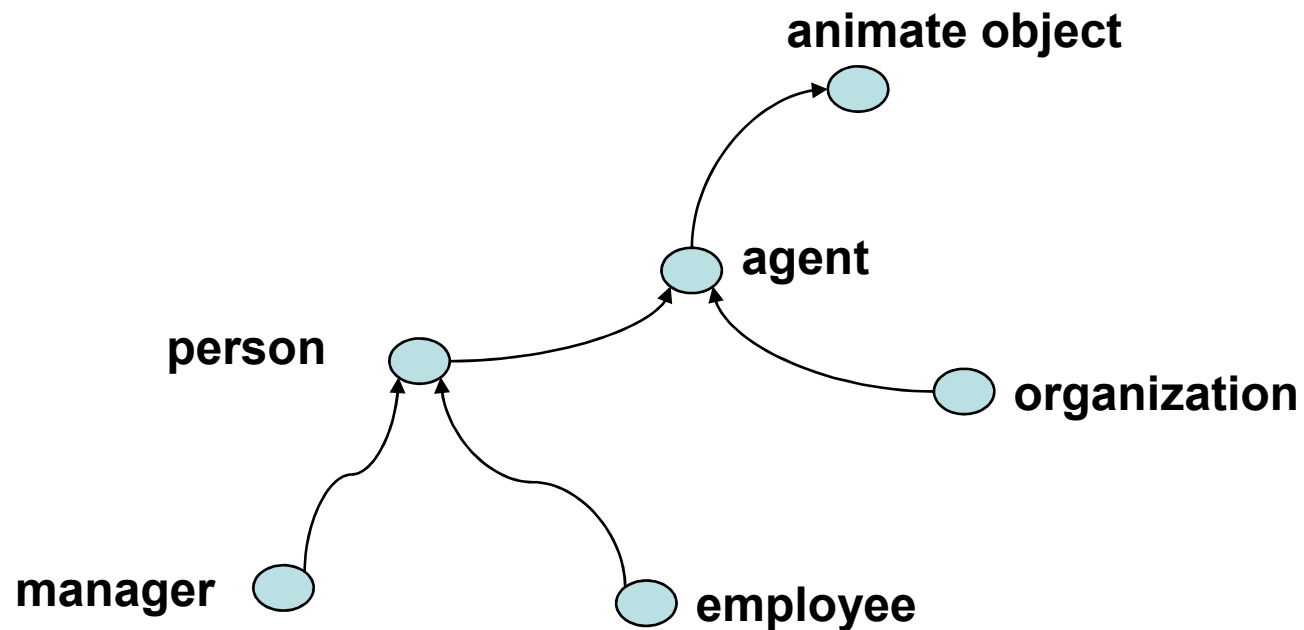
```
<?xml version="1.0"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>
<html xmlns=http://www.w3.org/1999/xhtml xml:lang="en" lang="en">
  <head>
    <title>Morning to-do list</title>
  </head>
  <body>
    <li>Wake up</li>
    <li>Make bed</li>
    <li>Drink coffee</li>
    <li>Go to work</li>
  </body>
</html>
```

Chapter 7: Understanding Taxonomies

- A taxonomy is a way of classifying or categorizing a set of things – specifically, a classification in the form of a hierarchy (a tree structure)
- An other definition of taxonomy:
 - The study of the general principles of scientific classification: orderly classification of plants and animals according to their presumed natural relationships
- The information technology definition for a taxonomy:
 - The classification of information entities in the form of a hierarchy, according to the presumed relationships of the real-world entities that they present

Taxonomies

- A taxonomy is usually depicted with the root of the taxonomy on top, as follows



Taxonomies

- Each node of the taxonomy (including root) is an information entity that stands for a real-world entity
- Each link between nodes represents a special relation
 - called the *is subclassification of* relation if the link's arrow is pointing up toward the parent node, or
 - called the *is superclassification of* if the link's arrow is pointing down at the child node
- When the information entities are classes these relations are defined more strictly as *is subclass of* and *is superclass of*

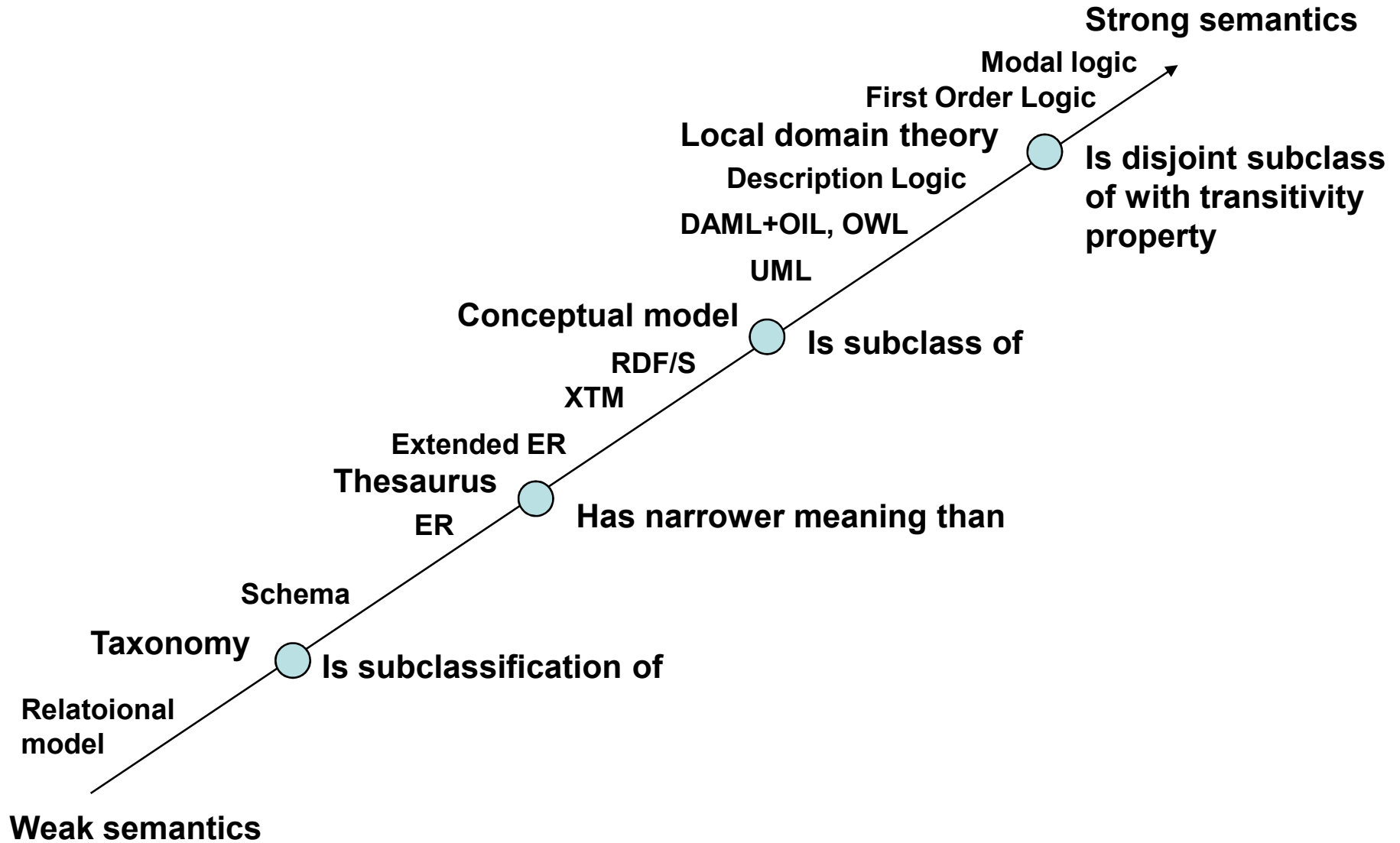
Taxonomies

- When one goes up the taxonomy towards the root, the entities become more general, and hence this kind of taxonomy is also called *generalization/specilization* taxonomy
- A taxonomy is a semantic hierarchy in which information entities are related by either the *subclassification of relation* or the *subclass of* relation
 - *subclassification of* is semantically weaker than *subclass of* relation, and so the difference between semantically stronger and weaker taxonomies can be done

The use of taxonomies

- A taxonomy is a way of structuring information entities and giving them a simple semantics
- On the web, taxonomies can be used to find products and services
 - E.g., UDDI has proposed the tModel as the placeholder for taxonomies such as UNSPSC and North American Industry Classification System that can be used to classify Web products and services
 - In addition the yellow pages of UDDI is a taxonomy, which is ordered alphabetically to be of additional assistance to a person looking for products and services

The ontology spectrum



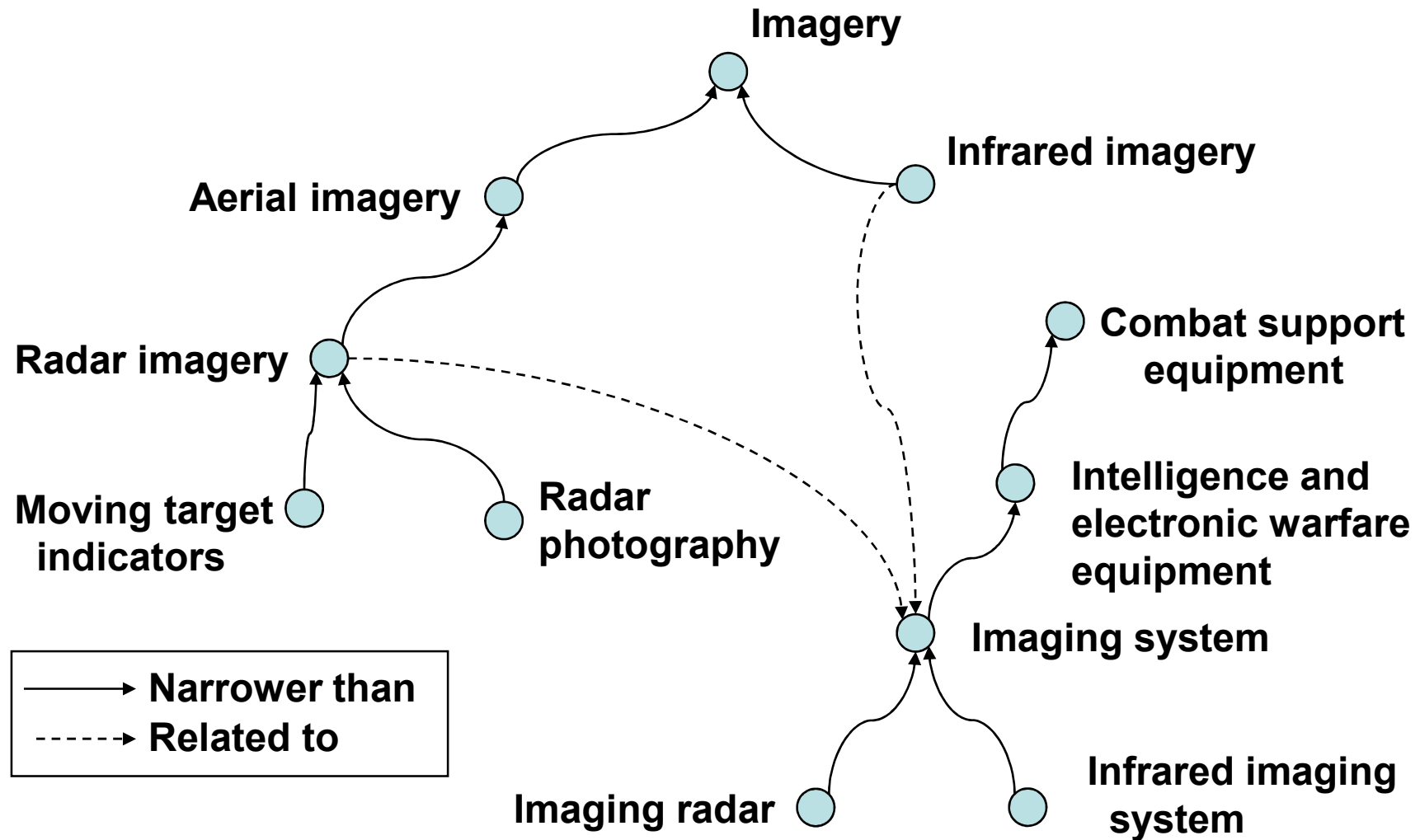
Thesarus

- Definition (ANSI/NISO Monolingual Thesarus Standard):
 - “a controlled vocabulary arranged in a known order and structured so that **equivalence, homographic, hierarchical,** and **associative** relationships among terms are displayed clearly and identified by standardized relationship indicators”
- The primary purposes of a thesarus are to facilitate retrieval of documents and to achieve consistency in the indexing of written or otherwise recorded documents and other items

Semantic Relations of a Thesarus

SEMANTIC RELATION	DEFINITION	EXAMPLE
Synonym Similar to Equivalent Used for	A term X has nearly the same meaning as a term Y	“Report” is synonym for “document.”
Homonym Spelled the same Homographic	A term X is spelled the same way as a term Y, which has a different meaning	The “tank”, which is a military vehicle, is a homonym for the “tank”, which is a receptacle for holding liquids
Broader Than (Hierarchic parent of)	A term X is broader in meaning than a term Y	“Organization” has a broader meaning than “financial institution”
Narrower Than (Hierarchic Child of)	A term X is narrower in meaning than a term Y	“Financial institution” has a narrower meaning than “organization”
Associated Associative Related	A term X is associated with a term Y, i.e., there is some unspecified relationship between the two	A “nail” is associated with a “hammer”

An example of a thesarus



Conceptual model

- A model of a subject area or area of knowledge, that represents entities, the relationships among entities, the attributes and, and sometimes rules
- Rules are typically of the following forms:
 - If X is true, then Y must also be true
 - I (W and X) or (Y and not Z) are true, then (U and V) must also be truewhere U, V, W, X,Y and Z are simple or complex assertions about the entities, relations or attributes
- If part of the rule is called *antecedent* while the then part is called *consequent*

Chapter 8: Understanding Ontologies

- Philosophical definitions
 - A particular theory about the nature of being or the kinds of existent
 - A branch of metaphysics concerned with the nature and relations of being
- Definitions from information engineering discipline point of view
 - Ontologies are about vocabularies and their meanings, with explicit, expressive and well-defined semantics, which is machine –interpretable
- Ontologies can be represented equally in a graphical and textual form
- Ontology languages are typically based on a particular logic, and so they are *logic-based languages*

Web Ontology language: OWL

- The expressivity of RDF and RDF Schema is deliberately very limited:
 - RDF is (roughly) limited to binary ground predicates
 - RDF Schema is (roughly) limited to a subclass hierarchy and a property hierarchy, with domain and range definitions of these properties
- However there are number of use cases that would require much more expressiveness than RDF and RDF Schema offer

Requirements for Ontology Languages

- Ontology languages allow users to write explicit, formal conceptualization of domain models. The requirements are:
 - A well defined syntax
 - A formal semantics
 - Convenience of expression
 - Efficient reasoning support
 - Sufficient expressive power
- OWL and DAML+OIL are build upon RDF and RDF Schema and have the same kind of syntax
 - Their syntax is not very user friendly, but on the other hand, users will develop their ontologies using ontology development tools

- **Formal semantics** describes the meaning of knowledge precisely, i.e., is not open to different interpretations by different people or machines
 - One use of a formal semantics is to allow people to reason about the knowledge. For ontological knowledge we may reason about:
 - **Class membership.** If x is an instance of class C , and C is a subclass of D , then we can infer that x is an instance of D
 - **Equivalence of classes.** If class A is equivalent to class B , and class B equivalent to class C , then A is equivalent to C , too.
 - **Consistency.** Suppose we have declared x to be an instance of the class A and A is a subclass of $(A \text{ intersection } B)$, A is a subclass of D , and B and D are disjoint. Then we have an inconsistency because A should be empty, but has the instance x . This is an indication of an error in the ontology.

- **Classification.** If we have declared that certain property-value pairs are a sufficient condition for membership in a class *A*, then if an individual *x* satisfies such conditions, we can conclude that *x* must be an instance of *A*
- Semantics is a prerequisite for *reasoning support*. *Reasoning support allows one to:*
 - Check the consistency of the ontology and the knowledge
 - Check for unintended relationships between classes
 - Automatically classify instances in classes
- *Automated reasoning support* allows one to check many more cases than could be checked manually. Checks like the preceding ones are valuable for designing large ontologies, where multiple authors are involved, and for integrating and sharing ontologies from various sources.

- A formal semantics and reasoning support are usually provided by mapping an ontology language to a known logical formalism, and by using automated *reasoners* that already exist for those formalisms

- OWL is (partially) mapped on a description logic, and makes use of existing *reasoners*, such as FACT and RACER
 - Description logics are a subset of predicate logic for which efficient reasoning support is possible

Limitations of the Expressive power of RDF Schema

- The main modeling primitives of RDF/RDFS concern the organization of vocabularies in typed hierarchies:
 - subclass and subproperty relationships,
 - domain and range restrictions,
 - instances and classes
- A number of features are missing including:
 - **Local scope of properties.** *rdfs:range* defines the range of a property, say *eats*, for all classes. Thus in RDF we cannot declare range restrictions that apply to some classes only.
 - For example, we cannot say that *cows* eat only plants, while other animals may eat meat, too

- **Disjointness of classes.** Sometimes it is useful to say that some classes are disjoint
 - For example, *male* and *female* are disjoint. In RDFS we can only state e.g., that *male* is a subclass of *person*
- **Boolean combination of classes.** Sometimes it is useful to build new classes by combining other classes using union, intersection, and complement
 - For example, we may define the class *person* to be the disjoint union of the classes *male* and *female*
- **Cardinality restrictions.** Sometimes it is useful to place restrictions on how many distinct values a property may or must take
 - For example, a person has exactly two parents, or that a course is lectured by at least one lecturer

- **Special characteristics of properties.** Sometimes it is useful to say that a property is
 - transitive (like “greater than),
 - unique (like “is mother of”), or
 - the inverse of another property (like “eats” and “is eaten by”)
- **Note.** The richer the ontology language is, the more inefficient the reasoning support becomes, often crossing the border of non-computability

Compatibility of OWL with RDF/RDFS

- **Ideally**, OWL would be an extension of RDF Schema, in the sense that OWL would use the RDF meaning of classes and properties (`rdfs:Class`, `rdfs:subclassOf`, etc.) and would add language primitives to support the richer expressiveness required
- **Unfortunately**, simply extending RDF Schema would work against obtaining expressive power and efficient reasoning because RDF Schema has some very powerful modeling primitives,
 - For example, constructions such as `rdfs:Class` (the class of all classes) and `rdf:Property` (the class of all properties) are very expressive and would lead to uncontrollable computational properties if the logic were extended with such expressive primitives

Three Species of OWL

- OWL Full
 - The entire language is called OWL Full and uses all the OWL languages primitives
 - The advantages of OWL full is that it is fully upward-compatible with RDF both syntactically and semantically:
 - any legal RDF document is also legal OWL Full document
 - any valid RDF/RDF Schema conclusion is also a valid OWL Full conclusion
 - The disadvantage of OWL Full is that the language has become so powerful as to be undecidable, dashing any hope of complete (or efficient) reasoning support

- OWL DL
 - In order to regain computational efficiency OWL DL (short for Description Logic) is a sublanguage of OWL Full that restricts how the constructors from OWL and RDF may be used
 - The disadvantage is that we lose full compatibility with RDF:
 - an RDF document will in general have to be extended in some ways and restricted in others before it is a legal OWL DL document
 - Every legal OWL DL document is a legal RDF document

- OWL Lite
 - An even further restriction limits OWL DL to a subset of the language constructors
 - For example, OWL Lite excludes enumerated classes, disjointness statements, and arbitrary cardinality
 - The advantages is that it is both easier to grasp (for users) and easier to implement (for tool builders)
 - The disadvantage is of course a restricted expressivity

- Ontology developers adopting OWL should consider which sublanguage best suits their needs
 - The choice between Lite and DL depends on the extent to which users require the more expressive constructs
 - The choice between OWL DL and OWL Full mainly depends on the extent to which users require the metamodeling facilities of RDF Schema (e.g., defining classes of classes, or attaching properties to classes)
 - When using OWL Full reasoning support is less predictable because complete OWL Full implementations will be impossible

- There are strict notions of upward compatibility between these three sublanguages:
 - Every legal OWL Lite ontology is a legal OWL DL ontology
 - Every legal OWL DL ontology is a legal OWL Full ontology
 - Every valid OWL Lite conclusion is a valid OWL DL conclusion
 - Every valid OWL DL conclusion is a valid OWL Full conclusion
- OWL still uses RDF and RDF Schema to a large extent:
 - All varieties of OWL use RDF for their syntax
 - Instances are declared as in RDF, using RDF descriptions and typing information
 - OWL constructors like `owl:Class` and `owl:DatatypeProperty`, and `owl:ObjectProperty` are specialisations of their RDF counterparts

- One of the main motivations behind the layered architecture of the Semantic Web is a hope of downward compatibility with corresponding reuse of software across various layers
- The advantage of full downward compatibility for OWL (that any OWL-aware processor will also provide correct interpretations of any RDF Schema document) is only achieved for OWL Full, at the cost of computational intractability

The OWL Language

- Syntax
 - OWL builds on RDF and RDF Schema and uses RDF's XML-based syntax
 - RDF/XML does not provide a very readable syntax and therefore also other syntactic forms for OWL have also been defined:
 - An xml-based syntax (<http://www.w3.org/TR/owl-syntax/>) that does not follow the RDF conventions and is thus more easily read by human users
 - An abstract syntax, used in the language specification document, that is much more compact and readable than XML syntax or the RDF/XML syntax
 - A graphic syntax based on the conventions of UML (Unified Modeling Language), which is widely used, and is thus an easy way for people to become familiar with OWL

Header

- OWL documents are usually called *OWL ontologies* and are RDF documents
- The root element of an OWL ontology is an *rdf:RDF element*, which also specifies a number of namespaces:

```
<rdf:RDF
  xmlns:owl=http://www.w3.org/2002/07/owl#
  xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:xsd=http://www.w3.org/2001/XMLSchema#>
```


- An OWL ontology may start with a collection of assertions for housekeeping.
- These assertions are grouped under an *owl:Ontology* element, which contains comments, version control, and inclusion of other ontologies. For example:

```
<owl:Ontology rdf:about="">
  <rdfs:comment>An example of OWL ontology</rdfs:comment>
  <owl:priorVersion
    rdf:resource=http://www.mydomain.org/uni-ns-old/>
  <owl:imports
    rdf:resource="http://www.mydomain.org/persons"/>
  <rdfs:label>University Ontology</rdfs:label>
</owl:ontology>
```

Note. *owl:imports* is a transitive property

Class Elements

- Classes are defined using an *owl:Class* element, e.g.,

```
<owl:Class rdf:ID="associateProfessor">  
  <rdfs:subClassOf rdf:resource="#academicStaffMember"/>  
</owl:Class>
```

- We can also say that this class is disjoint from other classes:

```
<owl:Class rdf:about="#associateProfessor">  
  <owl:disjointWith rdf:resource="#professor"/>  
  <owl:disjointWith rdf:resource="#assistantProfessor"/>  
</owl:Class>
```

- Equivalence of classes can be defined using an *owl:equivalentClass* element, e.g.,:

```
<owl:Class rdf:ID="faculty">  
  <owl:equivalentClass rdf:resource="#academicStaffMember"/>  
</owl:Class>
```

Property Elements

- In OWL there are two kind of properties:
 - **Object properties**, which relate objects to other objects, e.g., *isTaughtBy* and *supervises*
 - **Data type properties**, which relate objects to datatype values, e.g., *phone*, *title* and *age*.
 - OWL does not have any predefined data types, nor does it provide special definition facilities. Instead it allows one to use XML Schema data types, thus making use of the layered architecture of the Semantic Web
 - Example of datatype property:

```
<owl:DatatypeProperty rdf:ID="age">
  <rdfs:range
rdf:resource=http://www.w3.org/2001/XMLSchema#nonnegativeInteger/>
</owl:DatatypeProperty>
```
 - User-defined data types will usually be collected in an XML schema and then used in an OWL ontology

- Example of an object property:

```
<owl:ObjectProperty rdf:ID="isTaughtBy">  
  <rdfs:domain rdf:resource="#course"/>  
  <rdfs:range rdf:resource="#academicStaffMember">  
  <rdfs:subPropertyOf rdf:resource="#involves"/>  
</owl:ObjectProperty>
```

- OWL allows to specify "inverse properties", e.g., *isTaughtBy* and *Teaches*

```
<owl:ObjectProperty rdf:ID="teaches">  
  <rdfs:range rdf:resource="#course"/>  
  <rdfs:domain rdf:resource="#academicStaffMember">  
  <owl:inverseOf rdf:resource="#isTaughtBy"/>  
</owl:ObjectProperty>
```

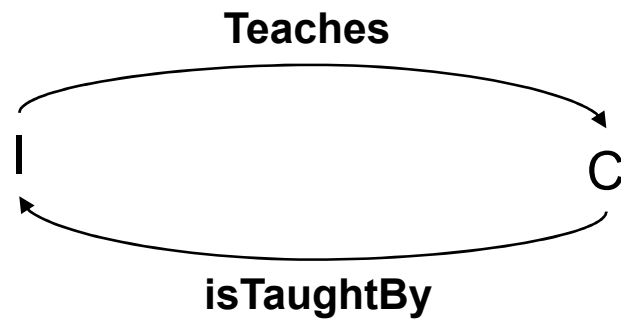


Figure. Relationship between a property and its inverse.

- Equivalence of properties can be defined through the use of the element `owl:equivalentProperty`

```
<owl:ObjectProperty rdf:ID="lecturesIn">  
  <owl:equivalentProperty rdf:resource="#teaches"/>  
</owl:ObjectProperty>
```

- There are also two predefined classes:
 - *Owl:Thing* is the most general class, which contains everything (everything is a thing)
 - *Owl:Nothing* is the empty class
- Thus every class is a subclass of *owl:Thing* and a superclass of *owl:Nothing*

Property Restriction

- **Example:** requiring first-year courses to be taught by professors only:

```
<owl:Class rdf:about="firstYearCourse">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource=#isTaughtBy"/>
      <owl:allValuesFrom rdf:resource=#Professor"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Note. This example requires *every* person who teaches an instance of the class, a first year course, to be professor. In terms of logic, we have a *universal quantification*.

- We can declare that mathematics courses are taught by David Billington:

```
<owl:Class rdf:about="mathCourse">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isTaughtBy"/>
      <owl:hasValue rdf:resource="#949352"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```


- We can declare that all academic staff members must teach at least one undergraduate course:

```
<owl:Class rdf:about="academicStaffMember">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource=#teaches"/>
      <owl:someValueFrom rdf:resource="#undergraduateCourse"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Note. This example requires that *there exists* an undergraduate course taught by an instance of the class, an academic staff member. It is still possible that the same academic teaches postgraduate courses in addition. In terms of logic, we have an *existential quantification*.

- In addition to restrictions *owl:allValuesFrom*, *owl:someValuesFrom* and *owl:someValuesFrom* we can set cardinality restrictions, e.g.,

```
<owl:Class rdf:about="#course">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isTaughtBy"/>
      <owl:minCardinality
        rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

- We might specify that a department must have at least ten and at most thirty members:

```
<owl:Class rdf:about="#department">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource=#hasMember"/>
      <owl:minCardinality
        rdf:datatype=&xsd;nonNegativeInteger">
        10
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource=#hasMember"/>
      <owl:maxCardinality
        rdf:datatype=&xsd;nonNegativeInteger">
        30
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Special Properties

- Some properties of property elements can be defined directly:
 - *Owl:TransitiveProperty* defines a transitive property, such as “has better grade than”, “is taller than” or “is ancestor of”
 - *Owl:SymmetricProperty* defines symmetric property, such as “has same grade as” or “is sibling of”
 - *Owl:FunctionalProperty* defines property that has at most one value for each object, such as “age”, “height”, or “directSupervisor”
 - *Owl:InverseFunctionalProperty* defines a property for which two different objects cannot have the same value, e.g., the property “isTheSocialSecurityNumber”

- An example of the syntactic forms for these is:

```
<owl:ObjectProperty rdf:ID="hasSameGradeAs">  
  <rdf:type rdf:resource="&owl;TransitiveProperty" />  
  <rdf:type rdf:resource="&owl;SymmetricProperty" />  
  <rdfs:domain rdf:resource="#student" />  
  <rdfs:range rdf:resource="#student" />  
</owl:ObjectProperty>
```

Boolean Combinations of classes

- We can specify union, intersection, and complement of classes
- For example, we can specify that courses and staff members are disjoint as follows:

```
<owl:Class rdf:about="course">  
  <rdfs:subClassOf>  
    <owl:Class>  
      <owl:complementOf rdf:resource=#staffMember"/>  
      <owl:someValueFrom rdf:resource="#undergraduateCourse"/>  
    </owl:Class>  
  </rdfs:subClassOf>  
</owl:Class>
```

This says that every course is an instance of the complement of staff members, that is, no course is a staff member

- The union of classes is built using *owl:unionOf*:

```
<owl:Class rdf:about="peopleAt Uni">  
  <owl:unionOf rdf:parseType="Collection">  
    <owl:Class rd:about="#staffMember"/>  
    <owl:Class rd:about="#student"/>  
  </owl:unionOf>  
</owl:Class>
```

Note. This does not say that the new class is a subclass of the union, but rather that the new class is *equal* to the union, i.e., an *equivalence of classes* is defined

- The intersection of classes is built using *owl:intersectionOf*:

```
<owl:Class rdf:id="facultyInCS">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rd:about="#faculty"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#belongsTo"/>
      <owl:hasValue rdf:resource="#CSDepartment"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```


Enumerations

- An enumeration is an *owl:oneOf* element, used to define class by listing all its elements:

```
<owl:Class rdf:ID="weekdays">
  <owl:one of rdf:parseType="Collection">
    <owl:Thing rdf:about="#Monday"/>
    <owl:Thing rdf:about="#Tuesday"/>
    <owl:Thing rdf:about="#Wednesday"/>
    <owl:Thing rdf:about="#Thursday"/>
    <owl:Thing rdf:about="#Friday"/>
    <owl:Thing rdf:about="#Saturday"/>
    <owl:Thing rdf:about="#Sunday"/>
  </owl:one of>
</owl:Class>
```

Instances

- Instances of classes are declared as in RDF:

```
<rdf:Description rdf:ID="949352">  
  <rdf:type rdf:resource="#academicStaffMember"/>  
</rdf:Description>
```

or equivalently

```
<academicStaffMember rdf:ID="949352">
```

further details can also be provided, e.g.,

```
<academicStaffMember rdf:ID="949352">  
  <uni:age rdf:datatype="&xsd;integer">39</uni:age>  
</academicStaffMember>
```

- Unlike typical database systems, OWL does not adopt the *unique-names assumption*
 - For example, if we state that each course is taught by at most one staff member

```
<owl:ObjectProperty rdf:ID="isTaughtBy">  
  <rdf:type rdf:resource="&owl;FunctionalProperty" />  
</owl:ObjectProperty>
```

and subsequently we can state that a given course is taught by two staff members

```
<course rdf:ID="CIT1111">  
  <isTaughtBy rdf:resource="#949318"/>  
  <isTaughtBy rdf:resource="#949352"/>  
</course>
```

This does not cause an OWL reasoner to flag an error

- To ensure that different individuals are different we have to state it, e.g.,

```
<lecturer rdf:ID="949318">  
  <owl:differentFrom rdf:resource="#949352"/>  
</lecturer>
```

- OWL also provides a shorthand notation to assert the pairwise inequality of all individuals in a given list

```
<owl:AllDifferent>  
  <owl:distinctMembers rdf:parseType="Collection">  
    <lecturer rdf:about="#94318"/>  
    .  
    .  
  </owl:distinctMembers>  
</owl:AllDifferent>
```

Data Types

- Although XML Schema provides a mechanism to construct user-defined data types (e.g., *adultAge* as integer greater than 18), such derived data types cannot be used in OWL.
- Not even all of the many built-in XML Schema data types can be used in OWL
- The OWL reference document lists all the XML Schema data types that can be used
 - These include the most frequently used types such as integer, Boolean, time and date

Layering OWL

- Now after the specification of the OWL we can specify which features can be used in its sublanguages OWL Full, OWL DL and OWL Lite
- **OWL Full**
 - All the language constructions may be used in any combination as long as the result is legal RDF

- **OWL DL**

- **Vocabulary partitioning**

- Any resource is allowed to be only a class, a data type, a datatype property, an object property, an individual, a data value, or part of the built-in vocabulary, and not more than one of these

- **Explicit typing**

- Not only must all resources be partitioned (as described in the previous constraint) but this partitioning must be stated explicitly

For example, if an ontology contains the following:

```
<owl:Class rdf:ID="C1">  
  <rdfs:subClassOf rdf:about="#C2" />  
</owl:Class>
```

Though this already entails that C2 is a class, this must be explicitly stated:

```
<owl:Class rdf:ID="C2"/>
```

– Property separation

- By virtue of the first constraint, the set of object properties and data type properties are disjoint. This implies that the following can never be specified for data type properties:

owl:InverseOf
owl:FunctionalProperty
Owl:InverseFunctionalProperty, and
Owl:SymmetricProperty

– No transitive cardinality restrictions

- No cardinality restrictions may be placed on transitive properties, or their subproperties, which are of course also transitive, by implication

– Restricted anonymous classes

- Anonymous classes are only allowed to occur as the domain and range of either *owl:equivalentClass* or *owl:disjointWith*, and as the range (but not the domain) of *rdfs:subClassOf*

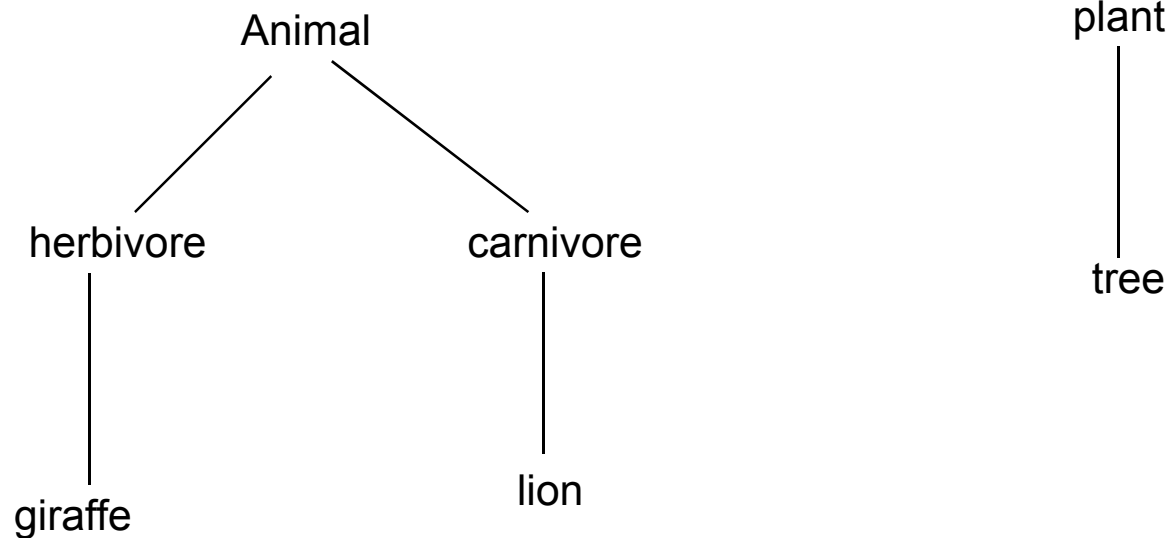
- **OWL Lite**

- An OWL Lite ontology must be an OWL DL ontology and must further satisfy the following constraints:

- The constructors *owl:one of*, *owl:disjointWith*, *owl:unionOf*, *owl:complementOf* and *owl:hasValue* are not allowed
 - Cardinality restrictions can only be made on the values 0 or 1
 - *Owl:equivalentClass* statements can no longer be made between anonymous classes but only between class identifiers

Example: An African Wildlife Ontology

- The ontology describes African wildlife. The above figure shows the basic classes and their subclass relationships. Note that the subclass information is only part of the information included in the ontology.



```
<rdf:RDF
  xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:owl=http://www.w3.org/2002/07/owl#
  <owl:Ontology rdf:about="xml:base"/>

  <owl:Class rdf:ID="animal">
    <rdfs:comment>Animals form a class.</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="plant">
    <rdfs:comment>
      Plants form a class disjoint from animals.
    </rdfs:comment>
    <owl:disjointWith rdf:resource="animal"/>
  </owl:Class>
```

```
<owl:Class rdf:ID="tree">  
  <rdfs:comment>Trees are a type of plant.</rdfs:comment>  
  <rdfs:subClassOf rdf:resource="#plant"/>  
</owl:Class>
```

```
<owl:Class rdf:ID="branch">  
  <rdfs:comment>Branches are part of trees.</rdfs:comment>  
  <rdfs:subClassOf>  
    <owl:Restriction>  
      <owl:onProperty rdf:resource="#is_part_of"/>  
      <owl:allValuesFrom rdf:resource="#tree"/>  
    </owl:Restriction>  
  </rdfs:subClassOf>  
</owl:Class>
```

```
<owl:Class rdf:ID="leaf">  
  <rdfs:comment>Leaves are part of branches.</rdfs:comment>  
  <rdfs:subClassOf>  
    <owl:Restriction>  
      <owl:onProperty rdf:resource="#is_part_of"/>  
      <owl:allValuesFrom rdf:resource="#branch"/>  
    </owl:Restriction>  
  </rdfs:subClassOf>  
</owl:Class>
```

```

<owl:Class rdf:ID="herbivore">
  <rdfs:comment>Herbivores are exactly those animals that eat only plants or part of
  plants.</rdfs:comment>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#animal"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eats"/>
      <owl:allValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#plant"/>
            <owl:Restriction>
              <owl:onProperty rdf:resource="#is_part_of"/>
              <owl:allValuesFrom rdf:resource="plant
            </owl:Restriction>
          </owl:unionOf >
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf >
</owl:Class>

```

```
<owl:Class rdf:ID="carnivore">
  <rdfs:comment>Carnivores are exactly those animals that eat
  animals.</rdfs:comment>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#animal"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eats"/>
      <owl:someValuesFrom rdf:resource="#animal"/>
    </owl:Restriction>
  </owl:intersectionOf >
</owl:Class>
```

```
<owl:Class rdf:ID="giraffe">
  <rdfs:comment>Giraffes are herbivores, and they eat only
  leaves.</rdfs:comment>
  <rdfs:subclassOf rdf:resource="#herbivore"/>
  <rdfs:subClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eats"/>
      <owl:allValuesFrom rdf:resource="#leaf"/>
    </owl:Restriction>
  </rdfs:subClassOf >
</owl:Class>
```



```
<owl:Class rdf:ID="lion">
  <rdfs:comment>Lions are animals that eat only
  herbivores.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#carnivore"/>
  <rdfs:subClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eats"/>
      <owl:allValuesFrom rdf:resource="#herbivore"/>
    </owl:Restriction>
  </rdfs:subClassOf >
</owl:Class>
```

```

<owl:Class rdf:ID="tasty_plant">
  <rdfs:comment>Tasty plants are plants that are eaten both by herbivores and
  carnivores.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#plant"/>
  <rdfs:subClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eaten_by"/>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#herbivores"/>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf >
  <rdfs:subClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#eaten_by"/>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#carnivores"/>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf >
</owl:Class>

```

```
<owl:TransitivityProperty rdf:ID="is_part_of"/>
```

```
<owl:ObjectProperty rdf:ID="eats">
```

```
  <rdfs:domain rdf:resource="#animal"/>
```

```
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="eaten:by">
```

```
  <owl:inverseOf rdf:resource="#eats"/>
```

```
</owl:ObjectProperty>
```

```
</rdf:RDF>
```

Chapter 9: Semantic Web Services

- **Web service**
 - Web site that do not merely provide static information, but involve interaction with users and often allow users to effect some action.
 - **Simple web service**
 - A single Web-accessible program, sensor or device that does not rely upon other Web services nor require further interaction with the user, beyond a simple request.
 - **Complex Web service**
 - Composed of simpler services, and often require ongoing interaction with the user, whereby the user can make choices or provide information conditionally,
 - e.g., searching CDs from online music store by various criteria, reading reviews and listening samples, adding CDs to shopping chart., providing credit card details, shipping details, and delivery address.

- *At present* the use of Web services requires human involvement, e.g., information has to be browsed and forms need to be filled in.
- The *Semantic Web vision*, as applied to Web services, aims at automating the discovery, invocation, composition and monitoring of Web services by providing machine-interpretable descriptions of the service.
 - Web sites should be able to employ a set of basic classes and properties by declaring and describing services, i.e., an ontology of services should be employed.
 - DAML-S is an initiative that is developing an ontology language for Web services. (Currently DAML-S is migrating to OWL-S).

- According to DAML-S there are three kinds of knowledge associated with a service: *service profile*, *service model* and *service groundings*.
- A ***service profile*** is a description of the offerings and requirements of a service.
 - This information is essential for a *service discovery*: service-seeking agent can determine whether a service is appropriate for its purpose, based on the service profile.
 - A service profile may also be a specification of a needed service provided by a service requester.
- A ***service model*** describes how a service works, i.e., what happens when the service is executed.
 - This information may be important for a service-seeking agent for composing services to perform a complex task, and monitoring the execution of a service.
- A ***service grounding*** specifies details of how an agent can access a service.
 - Typically a grounding will specify communication protocol and port numbers to be used in contacting a service.

Service Profiles (in DAML-S)

- Provide a way to describe services offered by a Web site but also services needed by requesters.
- Provide the following information:
 - A human-readable description of the service and its provider.
 - A specification of the functionalities provided by the service.
 - Additional information such as expected response time and geographic constraints.
- All this information is encoded in the modeling primitives of DAML-S (classes and properties, which are defined using the DAML-OIL language)

- For example, an offering of a service is an instance of the class OfferedService, which is defined as follows:
- ```
<rdfs: Class rdf: ID="OfferedService">
 <rdfs : label><OfferedService</rdfs :label>
 <rdfs:subClassOf rdf : resource= "http: //www.daml.org/services/daml-
 s/2001/10/Service.daml#(</>
</rdfs: class>
```

A number of properties are defined on this class:

- intendedPurpose
- serviceName
- providedBy
  - The range of this property is a new class ServiceProvider which has various properties. An instance of the class is the following:



```
<profile : ServiceProvider rdf : ID="SportNews">
 <profile :phone>12345678</profile :phone>
 <profile :fax>123456789</profile :fax>
 <profile :email>abc@defgh.com</profile :email>
 <profile :webURL>www.dfg.com</profile :webURL>
 <profile :PhysicalAddress>150 Nowwhere St,
 111Somewhere, Australia</profile :PhysicalAddress>
</profile : ServiceProvider>
```

- The functional description of a *service profile* defines properties describing the functionality provided by the service. The main properties are:
  - **Input**, which describes the parameters necessary for providing the service.
    - E.g., sports news service might require the following input: date, sports category, customer credit cards details.
  - **Output**, which specifies the outputs of the service.
    - E.g., in the sport news example, the output would be the news articles in the specified category at the given date.
  - **Precondition**, which specifies the conditions that need to hold for the service to be provided effectively.
    - E.g., credit card details are an input, and preconditions are that the credit card is valid and not overcharged.
  - **Effect**, a property that specifies the effects of the service.
    - E.g., credit card is charged \$1 per news article.

- At present the modeling primitives of DAML-S and OWL-S are very limited regarding the functional descriptions of the service because of limitations of the DAML+OIL (as well as OWL) language.
  - E.g., its is not possible to define logical relationships between inputs and outputs as the languages do not yet provide logical capabilities, e.g., rules.

# Service Models (in DAML-S)

- Service models are based on the concept of a **process**, which describes a service in terms of inputs, outputs, preconditions, effects and where appropriate, its composition of component subprocesses.
- *Class process* has three subclasses:
  - **Atomic processes**, can be directly invoked by passing them appropriate messages; they execute in one step.
  - **Simple processes** are elements of abstraction; they can be thought of as having single-step executions but are not invocable.
  - **Composite processes** consist of other, simpler processes.

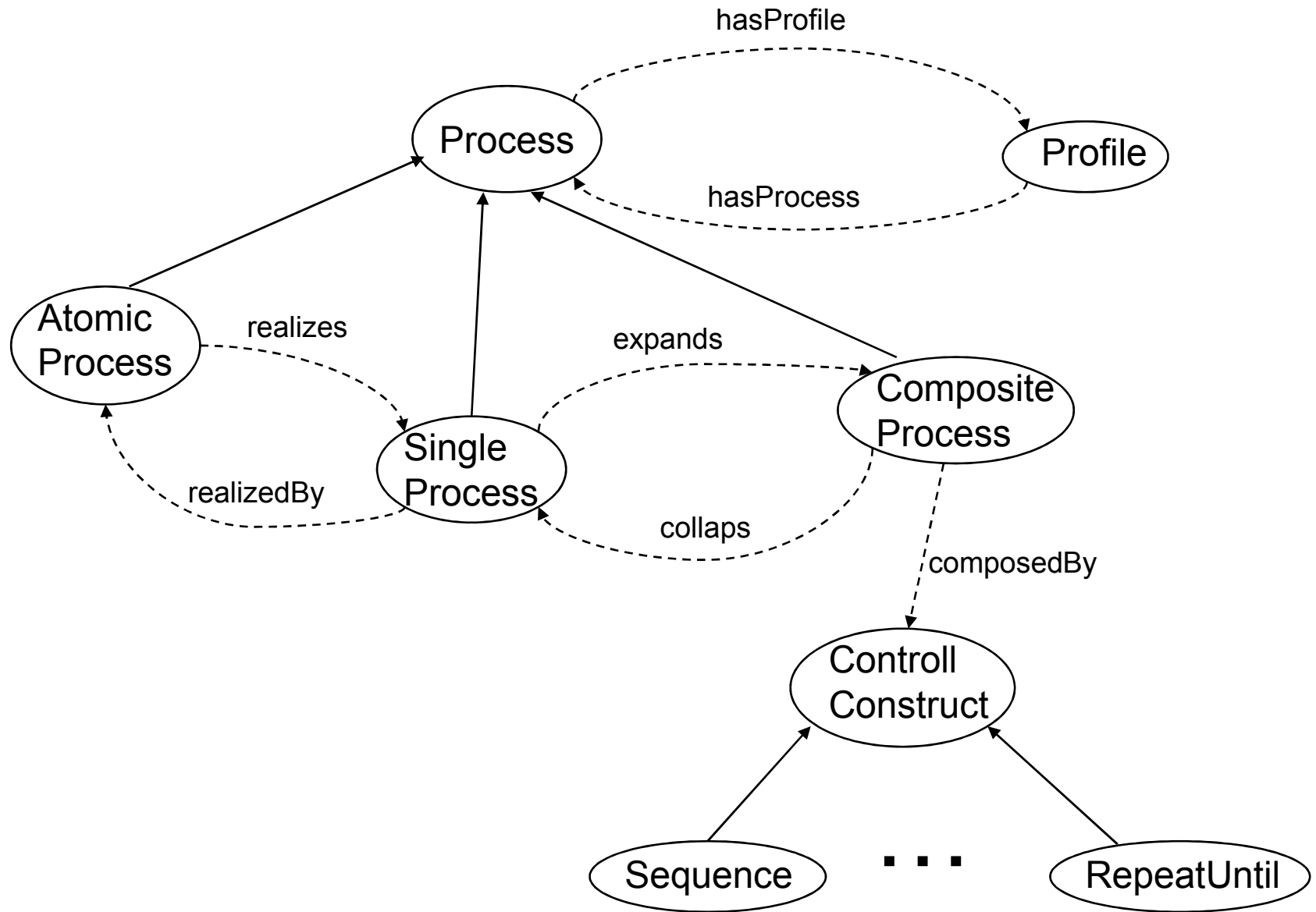


Figure. Top level of the process ontology

- In the figure,
  - hasProfile and hasProfile are two properties that state the relationship between a process and its profile.
  - A simple process may be realized by an atomic process.
  - Alternatively, it is used for abstraction purposes and *expands* to a composite process.
- A composite process is composed of a number of *control constructs*:

```

<rdf : Property rdf : ID="composedBy">
 <rdfs:domain rdf: resource="#CompositeProcess"/>
 <rdfs:range rdf : resource="#ControlConstruct"/>
</rdf : Property>

```

The control construct offered by DAML-S include, sequence, choice, if-then-else and repeat-until.

# **OWL-S: Semantic Markup for Web Services**

# The goal of OWL-S

- The Semantic Web should enable greater access not only to content but also to services on the Web:
  - Users and software agents should be able to discover, invoke, compose, and monitor Web resources offering particular services and having particular properties, and should be able to do so with a high degree of automation if desired.
- OWL-S (formerly DAML-S) is an ontology of services that makes these functionalities possible. It is comprised of three main parts:
  - the service *profile* for advertising and discovering services;
  - the *process model*, which gives a detailed description of a service's operation; and
  - the *grounding*, which provides details on how to interoperate with a service, via messages.
- Following the layered approach to markup language development, the current version of OWL-S builds on the OWL.



# Services on the Semantic Web

- Among the most important Web resources are those that provide services.
  - By "service" we mean Web sites that do not merely provide static information but allow one to effect some action or change in the world, such as the sale of a product or the control of a physical device.
- The Semantic Web should enable users to locate, select, employ, compose, and monitor Web-based services automatically.
- To make use of a Web service, a software agent needs a computer-interpretable description of the service, and the means by which it is accessed.
  - An important goal for Semantic Web markup languages, then, is to establish a framework within which these descriptions are made and shared.
  - Web sites should be able to employ a standard ontology, consisting of a set of basic classes and properties, for declaring and describing services, and the ontology structuring mechanisms of OWL provide an appropriate, Web-compatible representation language framework within which to do this.

- The OWL-S ontology is also referred as a *language* for describing services, reflecting the fact that it provides a standard vocabulary that can be used together with the other aspects of the OWL description language to create service descriptions.

# Motivating Tasks for OWL-S

- We will be considering both simple, or ``atomic" services, and complex or "composite" services.
- Atomic services are ones where a single Web-accessible computer program, sensor, or device is invoked by a request message, performs its task and perhaps produces a single response to the requester.
- With atomic services there is no ongoing interaction between the user and the service.
  - For example, a service that returns a postal code or the longitude and latitude when given an address would be in this category.

- Complex or 'composite' services are composed of multiple more primitive services, and may require an extended interaction or conversation between the requester and the set of services that are being utilized.
  - For example, one's interaction with [www.amazon.com](http://www.amazon.com) to buy a book is like this; the user searches for books by various criteria, perhaps reads reviews, may or may not decide to buy, and gives credit card and mailing information.
- OWL-S is meant to support both categories of services, but complex services have motivated many of the ontology's elements.
- The following three task types will give an idea of the kinds of tasks OWL-S is expected to enable

# Automatic Web service discovery

- Is an automated process for location of Web services that can provide a particular class of service capabilities, while adhering to some client-specified constraints.
  - For example, the user may want to find a service that sells airline tickets between two given cities and accepts a particular credit card.
- With OWL-S, the information necessary for Web service discovery could be specified as computer-interpretable semantic markup at the service Web sites, and a service registry or ontology-enhanced search engine could be used to locate the services automatically.
- Alternatively, a server could proactively advertise itself in OWL-S with a service registry, also called middle agent, so that requesters can find it when they query the registry.
  - Thus, OWL-S enables declarative advertisements of service properties and capabilities that can be used for automatic service discovery

# Automatic Web service invocation

- Automatic Web service invocation is the automatic invocation of an Web service by a computer program or agent, given only a declarative description of that service, as opposed to when the agent has been pre-programmed to be able to call that particular service.
- This is required, for example, so that a user can request the purchase, from a site found by searching and then selected by that user, of an airline ticket on a particular flight.
- Execution of a Web service can be thought of as a collection of remote procedure calls.

- OWL-S provides a declarative, computer-interpretable API that includes the semantics of the arguments to be specified when executing these calls, and the semantics of that is returned in messages when the services succeed or fail.
- A software agent should be able to interpret this markup to understand what input is necessary to invoke the service, and what information will be returned.
- OWL-S, in conjunction with domain ontologies specified in OWL, provides standard means of specifying declaratively APIs for Web services that enable this kind of automated Web service execution.

# Automatic Web service composition and interoperation

- Involves the automatic selection, composition, and interoperation of Web services to perform some complex task, given a high-level description of an objective.
  - For example, the user may want to make all the travel arrangements for a trip to a conference.
- Currently, the user must select the Web services, specify the composition manually, and make sure that any software needed for the interoperation of services that must share information is custom-created.



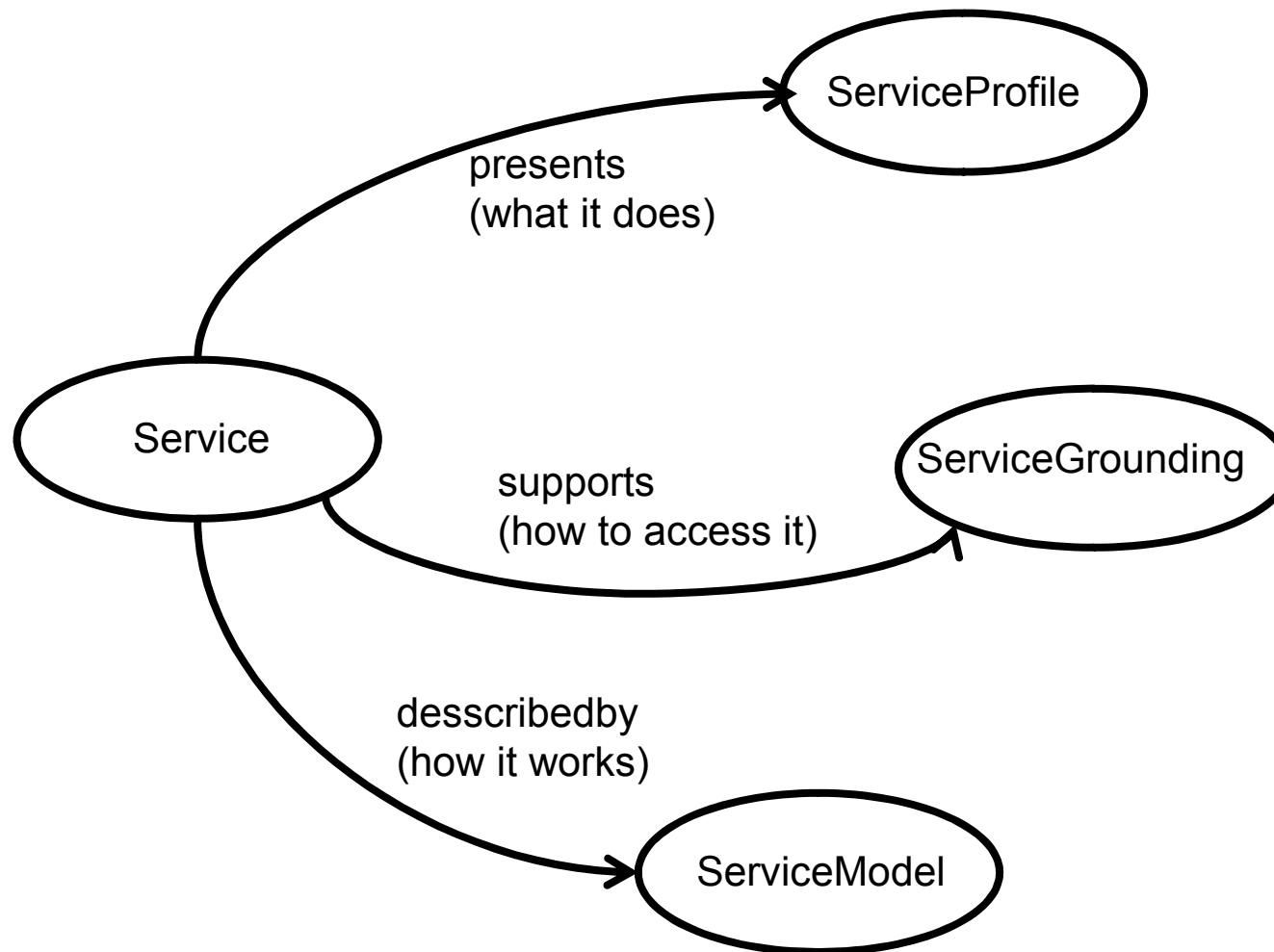
- With OWL-S markup of Web services, the information necessary to select and compose services will be encoded at the service Web sites.
- Software can be written to manipulate these representations, together with a specification of the objectives of the task, to achieve the task automatically.
- To support this, OWL-S provides declarative specifications of the prerequisites and consequences of application of individual services , and a language for describing service compositions and data flow interactions.

# An Upper Ontology for Services

The ontology of services is motivated by the need to provide three essential types of knowledge about a service :

- *What does the service provide for prospective clients?* The answer to this question is given in the "profile," which is used to advertise the service.
  - To capture this perspective, each instance of the class Service presents a ServiceProfile.
- *How is it used?* The answer to this question is given in the "process model." This perspective is captured by the ServiceModel class.
  - Instances of the class Service use the property describedBy to refer to the service's ServiceModel.
- *How does one interact with it?* The answer to this question is given in the "grounding." A grounding provides the needed details about transport protocols.
  - Instances of the class Service have a supports property referring to a ServiceGrounding.

# Top level of the service ontology



- The class `Service` provides an organizational point of reference for a declared Web service; one instance of `Service` will exist for each distinct published service.
- The properties *presents*, *describedBy*, and *supports* are properties of `Service`.
- The classes `ServiceProfile`, `ServiceModel`, and `ServiceGrounding` are the respective ranges of those properties.
- Each instance of `Service` will *present* a `ServiceProfile` description, be *describedBy* a `ServiceModel` description, and *support* a `ServiceGrounding` description.
  - The details of profiles, models, and groundings may vary widely from one type of service to another--that is, from one instance of `Service` to another.

# ServiceProfile

- Provides the information needed for an agent to discover a service, while the ServiceModel and ServiceGrounding, taken together, provide enough information for an agent to make use of a service, once found.
- Tells "what the service does", in a way that is suitable for a service-seeking agent (or matchmaking agent acting on behalf of a service-seeking agent) to determine whether the service meets its needs.
  - This form of representation includes a description of what is accomplished by the service, limitations on service applicability and quality of service, and requirements that the service requester must satisfy to use the service successfully.

# ServiceModel

- Tells a client how to use the service, by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and, where necessary, the step by step processes leading to those outcomes.
  - i.e., , it describes how to ask for the service and what happens when the service is carried out.
- For nontrivial services (those composed of several steps over time), this description may be used by a service-seeking agent in at least four different ways:
  - (1) to perform a more in-depth analysis of whether the service meets its needs;
  - (2) to compose service descriptions from multiple services to perform a specific task;
  - (3) during the course of the service enactment, to coordinate the activities of the different participants; and
  - (4) to monitor the execution of the service.

# ServiceGrounding

- A serviceGrounding ("grounding" for short) specifies the details of how an agent can access a service.
- Typically a grounding will specify a communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service. In addition, the grounding must specify, for each semantic type of input or output specified in the ServiceModel, an unambiguous way of exchanging data elements of that type with the service (that is, the serialization techniques employed).

- The upper ontology for services specifies only two cardinality constraints: a service can be described by at most one service model, and a grounding must be associated with exactly one service.
- The upper ontology deliberately does not specify any minimum cardinality for the properties *presents* or *describedBy*.
  - (Although, in principle, a service needs all three properties to be fully characterized, it is easy to imagine situations in which a partial characterization could be useful.)
  - Nor does the upper ontology specify any maximum cardinality for *presents* or *supports*. (It will be extremely useful for some services to offer multiple profiles and/or multiple groundings.)

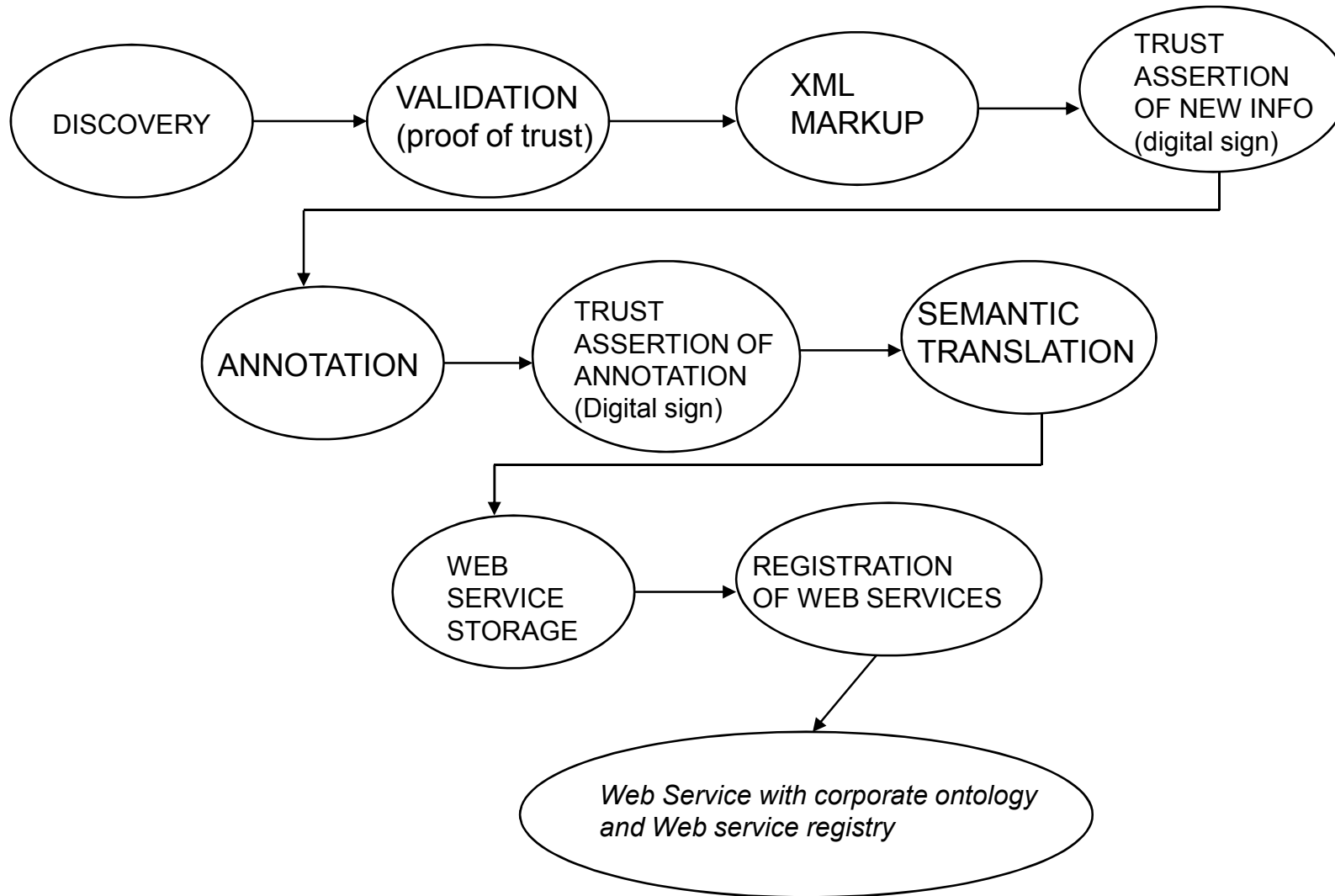


- Note that while we define one particular upper, one for service models, and one for ground ontology for profiles, nevertheless OWL-S allows for the construction of alternative approaches in each case.
- The intent here is *not* to prescribe a single approach in each of the three areas, but rather to provide default approaches that will be useful for the majority of cases.

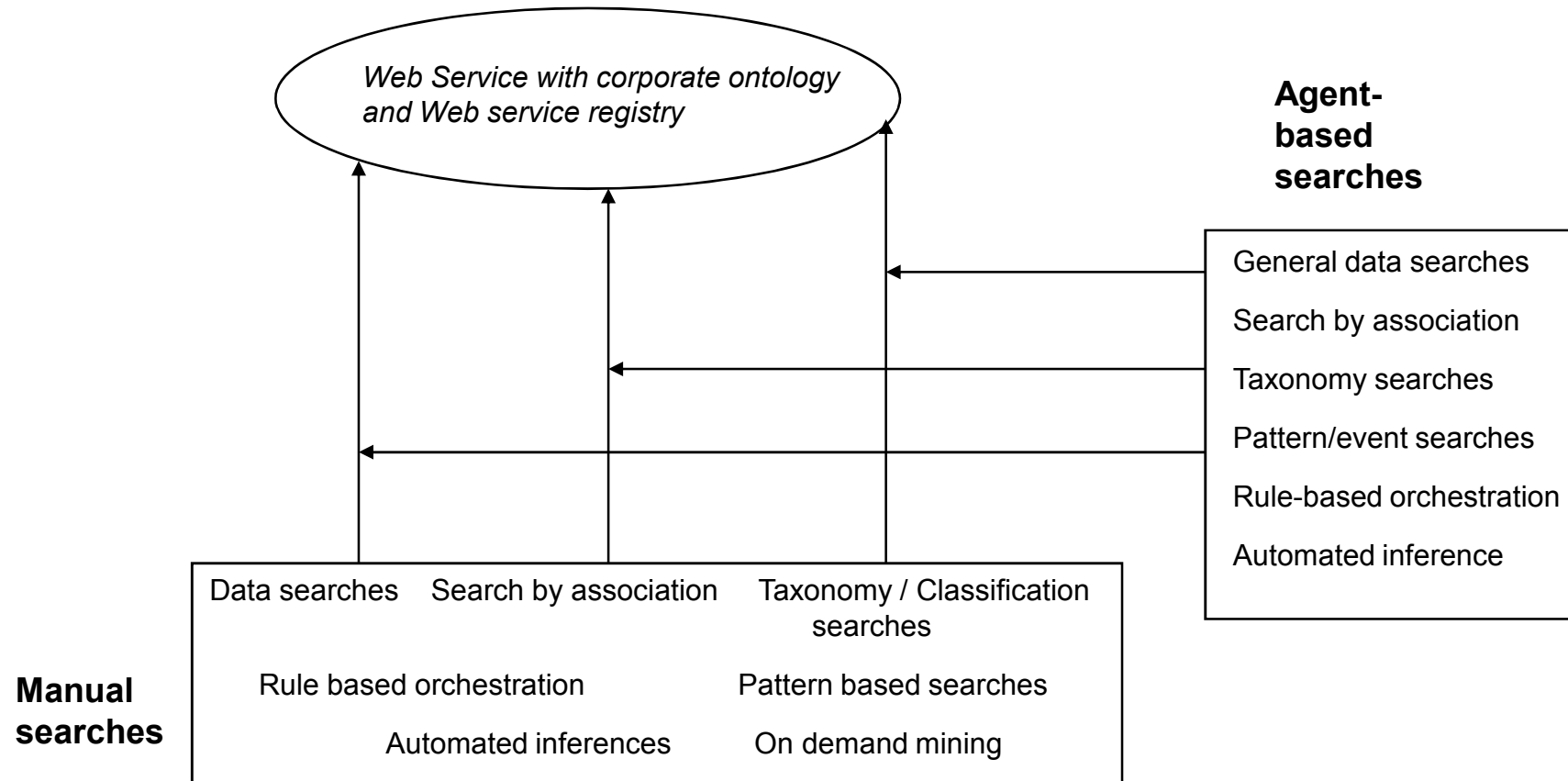
## Chapter 10: Organizations Roadmap to the Semantic Web

- Knowledge centric organization
  - A knowledge centric organization will incorporate Semantic Web technologies into every part of the work life cycle, including
    - Production
    - Presentation
    - Analysis
    - Dissemination
    - Archiving
    - Reuse
    - Annotation
    - Searches, and
    - Versioning

# The discovery and production process



# The search and retrieval process



## The functionality of the search and retrieval process

- Discovery of knowledge through taxonomy
  - Because each web service can be classified in various taxonomies, taxonomic searches can be done across the Web services of an organization, e.g., “I’m looking for all Web services classified in the corporate taxonomy as related to Coal mining’
- Web service-based data searches
  - Using standard SOAP interfaces, any application can query Web services in the enterprise
- Search by association
  - Because data is mapped into an ontology, semantic searches can be made across the entire knowledge base, e.g., “I would like to perform a query on all relatives of the terrorist Mohammed Atta, their closest friends, and their closest friends’ friends

## The functionality of the search and retrieval process, continues ...

- Pattern-based searches
  - Because all data can be semantically linked by relationships in the ontology, patterns that would only be seen in the past – by old data mining techniques that did not directly utilize meaning – can now be dynamically found with semantic searches, e.g., “Of all grocery stores listed in our corporate ontology, which stores have had revenue growth combined with an increased demand for orange juice”
- Manual and agent-based searches
  - Although all the searches can be manual, software agents can be equipped with rules to continually search the knowledge base and provide you with up-to-the-second result and alerts, e.g., “Alert me via email whenever a new document is registered discussing a new computer virus”

## The functionality of the search and retrieval process, continues ...

- Rule-based orchestration queries
  - Because Web services can be combined to provide modular functionality, rules can be used in order to combine various searches from different Web services to perform complicated tasks, e.g., Find me the lead engineer of the top-performing project in the company. Based on his favorite vacation spot from his response in his Human Resource survey, book him two tickets to that location next week, grant him vacation time, and cancel all of his work-related appointments”
- Automated inference support
  - Because the corporate ontology explicitly represents concepts and their relationships in a logical and machine-interpretable form, automated inference over the ontology and its knowledge becomes possible. Given a specific query, an ontology-based inference engine can perform deduction and other forms of automated reasoning to generate the possible implications of the query, thus returning much more meaningful results

# A strategy to take advantage of semantic Web technologies

- Set detailed technical goals
  - Markup documents in XML
  - Expose your applications as Web services
  - Build Web service orchestration tools
  - Establish a corporate registers
  - Build ontologies
  - Use tools that will help in production processes
  - Integrate search tools
  - Use an enterprise portal as a catalyst for knowledge engineering



## A strategy to take advantage of semantic Web technologies, continues ...

- Develop a plan with a workflow change strategy
- Set appropriate staff in place
- Set a schedule for implementing changes