Adaptive Reflective Software
Introduction

Compositional adaptation enables software to modify its structure and behaviour dynamically in response to changes in its execution environment.
Main Technologies Supporting Compositional adaptation

Separation of concerns → Computational adaptation
Computational reflection → Computational adaptation
Component based design → Computational adaptation

Middleware

That we consider as key to reconfigurable software design. Programmers can use these technologies to construct self-adaptive systems.
Separation of Concerns

Separation of concerns enables the separate development of an application’s functional behavior—that is, its business logic—and the code for Crosscutting concerns, such as quality of service (QoS), energy consumption, fault tolerance, and security.
Computational Reflection

Computational reflection refers to a program’s ability to reason about, and possibly alter, its own behavior.

Reflection enables a system to reveal selected details of its implementation without compromising portability.

Filip K McKinlej
Seyed Masoud Sadjadi
IEEE Computer 2004
Reflection

Reflection comprises two activities:

1. *Introspection* to let an application observe its own behavior, and

2. *Intercession* to let a system or application act on these observations and modify its own behavior.
Reflective System vs. Self-representation

A reflective system (represented as base-level objects) and its self-representation (represented as metalevel objects) are causally connected, meaning that modifications to either one will be reflected in the other.
MOP

A metaobject protocol (MOP) is an interface that enables “systematic” introspection and intercession of the base-level objects.

MOPs support either:

- structural reflection
- behavioral reflection
Metalevel understanding collected into metaobject protocols
Reflection Types

*Structural reflection* addresses issues related to class hierarchy, object interconnection, and data types. As an example, a metalevel object can examine a base-level object to determine what methods are available for invocation.

*Behavioral reflection* focuses on the application’s computational semantics. For instance, a distributed application can use behavioral reflection to select and load a communication protocol well suited to current network conditions.
Component-based Design

Popular component-based platforms include COM/DCOM, .NET, Enterprise Java Beans, and the Corba Component Model.

Component-based design supports two types of composition:

- **Static composition**, A developer can combine several components at compile time to produce an application.
- **Dynamic composition**, the developer can add, remove, or reconfigure components within an application at runtime.
  
  Late binding
Recomposition Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function pointers</td>
<td>Application execution path is dynamically redirected through modification of function pointers.</td>
<td>Vtables in COM, delegates and events in .NET, callback functions in Corba</td>
</tr>
<tr>
<td>Wrappers</td>
<td>Objects are subclassed or encapsulated by other objects (wrappers), enabling the wrapper to control method execution.</td>
<td>ACE, R-Java, PCL, QuO, TRAP/J</td>
</tr>
<tr>
<td>Proxies</td>
<td>Surrogates (proxies) are used in place of objects, enabling the surrogate to redirect method calls to different object implementations.</td>
<td>ACT, AspectIX</td>
</tr>
<tr>
<td>Strategy pattern</td>
<td>Each algorithm implementation is encapsulated, enabling transparent replacement of one implementation with another.</td>
<td>DynamicTAO and UIC</td>
</tr>
<tr>
<td>Virtual component pattern</td>
<td>Component placeholders (virtual components) are inserted into the object graph and replaced as needed during program execution.</td>
<td>ACE and TAO</td>
</tr>
<tr>
<td>Metaobject protocol</td>
<td>Mechanisms supporting intercession and introspection enable modification of program behavior.</td>
<td>Open Java, Kava, TRAP/J, Open ORB, Open COM, Iguana/J</td>
</tr>
<tr>
<td>Aspect weaving</td>
<td>Code fragments (aspects) that implement a crosscutting concern are woven into an application dynamically.</td>
<td>AspectJ, Composition Filters, TRAP/J, AspectIX, Iguana/J, Prose</td>
</tr>
<tr>
<td>Middleware interception</td>
<td>Method calls and responses passing through a middleware layer are intercepted and redirected.</td>
<td>ACT, IRL, Prose</td>
</tr>
<tr>
<td>Integrated middleware</td>
<td>An application makes explicit calls to adaptive services provided by a middleware layer.</td>
<td>Adaptive Java, Orbix, Orbix/E, ORBacus, BBS, CIAO, Iguana/J, Ensemble</td>
</tr>
</tbody>
</table>
COMPOSITIONAL ADAPTATION TAXONOMY

How

When

Where

What

software composition takes place?
ADAPTATION TAXONOMY
How to Compose

All of the techniques in Table create a level of indirection in the interactions between program entities.

Some techniques use specific software design patterns to realize this indirection, whereas others use AOP, reflection, or both.

The two middleware techniques both modify interaction between the application and middleware services, but they differ in the following way: Middleware interception is not visible to the application, whereas integrated middleware provides adaptive services invoked explicitly by the application.
How to Compose

*composer* - the entity that uses some techniques to adapt an application.

The composer might be a human—a software developer or an administrator interacting with a running program through a graphical user interface—or a piece of software—an aspect weaver etc.

When and where the composer modifies the program determines the *transparency* of the recomposition.

Transparency refers to whether an application or system is aware of the “infrastructure” needed for recomposition.
When
Wehre

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-specific middleware services</td>
</tr>
<tr>
<td>Common middleware services</td>
</tr>
<tr>
<td>Distribution middleware</td>
</tr>
<tr>
<td>Host-infrastructure middleware</td>
</tr>
<tr>
<td>Operating systems and protocols</td>
</tr>
<tr>
<td>Hardware devices</td>
</tr>
</tbody>
</table>

Four-layer decomposition of middleware to bridge the gap between an application program and the underlying operating systems, network protocols, and hardware devices.
CORBA Example

Client application

Client

Applications

Domain services

Common services

Distribution

Host infrastructure

System platform

Server application

Servant

Skeleton

Server ORB

Network

Request flow

Reply flow
What

Category of Reflection

*Introspection*: The system structure can be accessed but not modified. If we take Java as an example, with its java.lang.reflect package, we can get information about classes, objects, methods and fields at runtime.

*Structural Reflection*: The system structure can be dynamically modified. An example of this kind of reflection is the addition of object’s fields.

*Computational (Behavioral) Reflection*: The system semantics (behavior) can be modified. In the standard Java API v.1.3, the class java.lang.reflect.Proxy has been added; it can be used to modify the dispatching method mechanism, being handled by a proxy object.

*Linguistic Reflection*. As an example, with OpenJava reflective language, the own language can be enhanced to be adapted to specific design patterns.
Reflective Process

The key to the approach is to make some aspects of the internal representation of the middleware explicit, and hence accessible from the application, through a process called reification.

The process where some aspects of the system are altered or overridden is called absorption.

The key to the approach is to make some aspects of the internal representation of the middleware explicit, and hence accessible from the application, through a process called reification.
Behavior of Middleware

The behaviour of the middleware with respect to a specific application is described as a set of associations between:

• the *services* that the middleware customises,
• the *policies* that can be applied to deliver the services, and
• the *context configurations* that must hold in order for a policy to be applied.
Role and Responsibility of Reflective Process

Our model assumes that the behaviour of the middleware with respect to a particular service is determined, at any time, by one and only one policy.

Process steps

Middleware service invocation → Middleware checks app. profile → Resource status query → Decision which policy to apply in current context
Application Profile Abstract Syntax

\[
\begin{align*}
\text{serviceList} & : = \text{service serviceList} | \varepsilon \\
\text{service} & : = \text{sname policyList} \\
\text{policyList} & : = \text{policy policyList} | \text{policy} \\
\text{policy} & : = \text{pname contextList} \\
\text{contextList} & : = \text{context contextList} | \text{context} \\
\text{context} & : = \text{resourceList} \\
\text{resourceList} & : = \text{resource resourceList} | \varepsilon \\
\text{resource} & : = \text{rname oname valueList} \\
\text{valueList} & : = \text{value valueList} | \varepsilon
\end{align*}
\]
Customisation of the Messaging Service
Example

- MessagingService
  - plainMsg
    - bandwidth > 40%
- compressedMsg
  - bandwidth < 40%
- policy
- context
Reflective Design

*Reflective Designs* enhance object-oriented design and programming by techniques for runtime system adaptation.

There are two key notions:

*design elements*, and

*design operators*.

Robert Hirschfeld,  
Ralf Lammel  
DoCoMo Communications  
Laboratories Europe
Design Elements

We use the term design element to denote representations of design decisions in programs.

In fact, we require that design elements are amenable to reflection such that design decisions can be observed and modified at runtime.
Design Operators

When compared to basic techniques such as the use of a metaobject protocol, the use of design elements makes runtime system adaptations more disciplined and more manageable.

To this end, we provide abstractions that capture common design elements in a reusable manner. Applications of such abstractions perform system adaptations at a design level; hence, we call them design operators.

*Operators that model the realisation of common design patterns at run-time.*

*Squeak/Smalltalk – implementation of Reflective Designs*
Examples

Reflective Middleware for Integrating Network Monitoring with Adaptive Object Messaging
Mission Statement

In the future, applications will need to execute in a ubiquitous environment with varying network conditions (connectivity, bandwidth, etc.) and system constraints (e.g., power and storage). The distributed object paradigm is often used to facilitate the development of large-scale distributed applications.

However, the traditional object messaging layer operates with limited awareness of underlying system and network conditions, whereas current system and network monitoring tools operate at the network layer with little awareness of application-level object communication requirements.
A reflective architecture for bridging system monitoring middleware and reflective object messaging.
A reflective architecture for bridging system monitoring middleware and reflective object messaging
Autonomic Computing
AC & Self-Management

• **Self-configuring**
  – auto-adaptation zmian w systemie

• **Self-healing**
  – wykrywanie lub predykcja nieprawidłowych operacji i podejmowanie czynności zaradczych (notyfikacja elementu, który wywołał błąd, a także pozostałych elementów systemu) celem usunięcia problemu

• **Self-optimizing**
  – efektywna alokacja zasobów (near term) i load-balancing
  – “learn from experience” w celu dopasowania idealnych parametrów systemu (long term tune)

• **Self-Protecting**
  – dostarczenie właściwych informacji do właściwych użytkowników, we właściwym czasie i używając procedur dopasowanych do roli użytkownika
  – walka z overloadingiem systemu
  – walka z denial-of-service attacks
AC Levels

- Basic – manual analysis and problem solving
- Managed – centralized tools, manual actions
- Predictive – cross-resource correlation and guidance
  - Adaptive – system monitors, correlates and takes action
  - Autonomic – dynamic business-policy based management
IBM – The 8 elements of AC

- needs to “know itself”: current system status, elements (also that one to be “borrowed”), active connections
- must configure/reconfigure itself under varying conditions
- “never settles for status quo” - always look for ways to optimize its workings
- must be able to recover from malfunctions
- a virtual world is no less dangerous than physical one
- must know its environment and context to best interact with neighbours
- must live in a heterogenous world and implement open standards
- must anticipate resource usage (hidding its complexity from user)
AC Architecture

1. **MONITOR**: look for events (detected by sensors)
   use knowledge base to understand what you are looking at
2. **ANALYZE**: use knowledge to determine what to do with event
3. **PLAN**: after event analysis determine what the action to take
   (formulate the plan)
4. **EXECUTE**: use effectors to execute the action(s)
Resource management aspects

Policy-based Management

IBM model

JIMS instrumented resources
Resource management aspects

Motorola Model

AC – still concept far away for us

Practical approach: RM-API and PMAC
RM API – JSR 284

- Resource Consumption Management API
- Resource - CPU, open sockets, network etc.
  - unit, name, granularity, measurement delay
- Resource domain – encapsulates resource policy logic
  - consume/unconsume, set reservation
  - set/remove consume actions
- Consume action
  - pre/post consume callback
  - trigger
  - defines how the policy is implemented
Application Isolation API

• It related to MVM concept
• Provides a uniform mechanism for managing Java application life cycles that are isolated from each other
• Isolate – a “handle” to an isolated computation
• Isolates cannot detect the presence of each other except through use of the Isolation API
• RM API + Isolation API
  – different isolates can be bound to different resource domains and use different policies
Goals of TRM system

• **Transparent Resource Management**
  – framework
  – MVM implementation

• **Transparency** – application developer is not involved

• Separation between business and resource management logic

• Application source code stays unmodified

• RM logic described as an **adaptability strategy**

• Different strategies can be applied - reusability

**Separation of Concerns**
Adaptability strategy in TRM

- Finite State Machine
- Each state defines policy (as constraints) for resources
  - initial
  - active
- Conditions defines when the current policy is changed
- Constraints and conditions
  - network bandwidth
  - CPU usage, number of files etc.
Application deployment in TRM

Adaptation Strategy Definition -> Adaptation Strategy Specification -> RM Awareness Engine

RM Unaware Application -> RM API App elements

RM Aware Application -> Monitoring API App elements

TRM Execution Environment
Main elements of TRM execution environment

- **Manager (per VM)**
  - interact with MSA
  - implement FSM logic
  - start/stop/resume/pause application
  - interact with other managers

  map application id onto TRM node id

- **Monitoring agent (per VM)**
  monitors non-RM API conditions
Two layers in TRM system

State
Set of Resource Domains
Policy (consume actions, triggers)

Application Layer
Resource Domains Layer

State A -> State B
State B -> State C
State C

TRM API

RM API

Resource Domains for state A
Resource Domains for state B
Resource Domains for state C
Experimental results - MVM

- Multithread “computational” application
  - each thread is very time-consuming (up to 100% CPU)
  - new thread is started every one second
- AS applied to application
  - condition – no. of threads
  - constraint – max. CPU usage
- Graph of states
  - three states defined
  - simple, “linear” (for clarity)

Virtualization level = Process = MVM (or cluster of MVMs)
Experimental results

<table>
<thead>
<tr>
<th>Policy</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>10%</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>II</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>II</td>
<td>5%</td>
<td>80%</td>
<td>5%</td>
</tr>
</tbody>
</table>
Bandwidth policy example

- **Initial state**: 10s 10s 10s 10s 10s
- **Bandwidths**:
  - A: 5kb/s
  - B: 13kb/s
  - C: 20kb/s
  - D: 25kb/s
  - E: 37kb/s

- Graph showing bandwidth over time.
Experimental results – Zone, Task, Project

JMX – based Impl. of RM API

JIMS for Solaris10
RM-API JMX based implementation
Resource allocation strategy
CPU-shares allocations

**Policy:** Periodically

- Project 1 – 80
- Project 2 – 20 – 80/20
- Project 1 – 20
- Project 2 – 80 – 20/80
Retention problem

Policy: Periodically
Project 1 – 80 Project 2 – 20
Project 1 – 20 Project 2 – 80

Limit violation
Adjusting Resources to meet QoS specification

Applications response time
CPU shares assigned to projects
Conclusions

- The existing resource virtualization technology provides powerful mechanisms for RM-control.
- It is possible to construct adaptive and reflective middleware which could exploit RM-control potential in transparent way.
- Much more is needed for AC.
- Other work at AGH:
  - PMAC- IBM model exploitation
  - Adaptability of CCM application
MDA
Model Driven Architecture
What’s the problem?

• Software is expensive, and productivity is low for many reasons. Amongst them:
  – Code is at too low level of abstraction
  – Reuse occurs (to the extent it does at all) at too low a granularity
  – Any code is glued together (at great expense) to its infrastructure (also expressed as code)
  – Mapping information (design expertise) is applied—then lost

No wonder!

Expensive and hard-to-find!
Why make models?

• Two main challenges for software development
  – Master complexity
  – Manage change

• Models meet these by
  – Acting as a means of communication between stakeholders
  – Dealing with varying levels of detail
    • Abstraction to hide details
      – isolate changes at lower level of abstraction
    • Refinement to introduce details
  – Being subject of analysis and other automatic tasks
Why model?

• A good model:
  – Abstracts away not-currently-relevant stuff
  – Accurately reflects the relevant stuff, so it…
  – Helps us reason about our problem
  – Is cheaper to build than code
  – Communicates with people
  – Communicates with machines
What is a model?

• A model is coherent set of elements that:
  – Covers some subject matters
    • Doesn’t have to cover all subject matters
  – At some level of abstraction
    • Doesn’t have to define realizations
  – That need not expose everything
    • Doesn’t have to show everything at once
  – That need not be complete in itself
    • Doesn’t have to include “code”

Principles, standards and tools

Model-Driven Engineering (MDE)

Principles
- MDA™
- MIC

Standards
- Model-Driven Architecture (OMG)
- Model Integrated Computing
- Software Factories (MS)
- Other Standards

Tools
- Eclipse
- EMF
- GMF
- GME
- Microsoft Visual Studio Team system DSL Tools
- Other Tools
MDA From 30.000 Feet

- A PIM can be retargeted to different platforms
- The concept Platform is relative
  - A model can be classified as both PIM and PSM
  - Tip: use concrete platform to qualify instead of “PIM” and “PSM”
    - “Component-architecture independent model” instead of PIM, “J2EE-specific model”, “.Net-specific model” instead of PSM
Components of an MDA solution

- Capture each layer in a platform-independent manner as intellectual property.
- Capture the mappings to the implementation as intellectual property (IP).
- Models and mappings become assets.

Layer by layer.
Enter Model-Driven Architecture

• MDA is three things:
  – An OMG initiative to promote model-driven development
  – A brand for standards and conforming products
  – A set of technologies and techniques

© OMG

Our focus
Enter Model-Driven Architecture

• MDA:
  – Captures IP as models and enables protection of them
  – Allows IP to be mapped automatically
  – Allows multiple implementations
  – Makes IP portable

This enables a market for IP in software.
**Instance-of**

**Real World**
- slug
- stray
- feral

**Models**
- Fido (20Kg, Awful): Dog
- Munchin (16Kg, FeedingOnly): Cat
- LapKitty (12Kg, LapLover): Cat

**Entity classifications**
- feral
- slug
- stray

**Class Model**
- Dog + slobberFactor
- Cat + standOffIndex

**Instance Model**
- Pet + name + weight

**Abstract**
- Reflects Real World Models
The relationship to the metamodel

Problem domain: A modeling language (i.e. a Metamodel)

Entity classifications

Problem domain: A model

Class Model

Class

Attribute

Instance of

Pet
Dog
Cat

Pet::name
Pet::weight
Dog::slobberFactor
Cat::standOffIndex

Abstract

Classify

Abstract

Classify

Abstract

Classify

Abstract

Classify

Abstract

Classify

Abstract

Classify

Abstract

Classify
# Four Layer Metamodeling Architecture

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta-metamodel</td>
<td>The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.</td>
<td>MetaClass, MetaAttribute, MetaOperation</td>
</tr>
<tr>
<td>metamodel</td>
<td>An instance of a metamodel. Defines the language for specifying a model.</td>
<td>Class, Attribute, Operation, Component</td>
</tr>
<tr>
<td>model</td>
<td>An instance of a metamodel. Defines a language to describe an information domain.</td>
<td>StockShare, askPrice, sellLimitOrder, StockQuoteServer</td>
</tr>
<tr>
<td>user objects (user data)</td>
<td>An instance of a model. Defines a specific information domain.</td>
<td>&lt;Acme_SW_Share_98789&gt;, 654.56, sell_limit_order, &lt;Stock_Qoute_Svr_32123&gt;</td>
</tr>
</tbody>
</table>
Focusing on The Business of IT Architecture

Stable evolution of your business designs
Convergent MDA mapping for business continuity.

The Business of high-ROI Business Solutions

Stable evolution of your UML models
Convergent MDA mapping for business continuity.

Managed Evolution of Service Infrastructure:
EAI, framework standards, custom
e.g. J2EE/EJB, .NET, Host, SAP
Optimized mapping for effective usage and ROI.

BEA, IBM, Borland, IONA, MSoft, …
The first level of automation ~30 Last years

Programming IDEs (e.g. JBuilder, Visual Age, NetBeans)

Environment

PIM

Programming Language

• Higher level of expression
• Easier to understand
• Portable
• Standardized

Generator, Projection

Compiler Engine

• Dependable
• Flexible
• Configurable
• Optimizing
• Complete: Linker, Debugger, Etc.

PSM

Diverse HW/OS Platforms

1st P-stack
MDA = New levels of automation ~ Last 8 Years

Architectural IDEs

Environment

PIM Model (UML, BOM...) &
Modeling Style (J2EE, eEPC, COBOL, .NET...)

• Higher level of expression
• Easier to understand
• Portable
• Standardized

Generator, Projection

Translative Generator Engine

• Dependable
• Flexible
• Configurable
• Debuggable
• Optimizing
• Complete

Models to Code
Models to Models

P-Stack: A Level of Automation

1st
MDA Convergence.

Automation Level
(P-Stack)

4th Business Object Model → Pattern based refinement module
Model to Model → 2 tier UML model

3rd Two tier UML model → Internet Accessor generator
Model to Model → UML model of all 4 tiers

2nd Four tier UML model → Translative generator w/ J2EE Cartridge.
Model to Code → Tuned 4 tier J2EE/EJB environment

1st Four tier J2EE/EJB environment → Compile, build, test using the model generated env.
Code to Binary → Operative 4 tier J2EE systems
Managing Corporate Architectural Style

The Unified Process

ArcStyler Core Modules

Open UML/XML (XMI) Repository

- Business Object Modeler
- Pattern Refinement Assistant
- UML Refinement Assistant
- Generator Engine with Meta IDE

MDA Cartridges

- Build, Deploy & Test Support

Optional Integrated Tools

- IDS ARIS
- Rational Rose
- Java IDE

Architectural Style Builder & MDA Programming IDE

Std. MDA Projections

- J2EE/EJB, .NET
- BEA WebLogic
- IBM WAS NT, z/OS
- Borland BES
- IONA iPAS
- Custom Infrastr: CORBA, Host...
Models versus Ontologies - What's the Difference and where does it Matter?
Brief History

- Ontologies
  - originated from the artificial intelligence world for the purpose of precisely capturing “knowledge”
  - used under an Open World Assumption (OWA)
    - new “knowledge” discovered by automated reasoning
  - characterized by OWL as the flagship language
    - formal semantics (description logic)

- Models (à la MDA)
  - originated from the software engineering world for the purpose of simplifying the description of software
  - used under a Closed World Assumption (CWA)
    - information defined prescriptively for construction
  - characterized by UML as the flagship language
    - semi-formal semantics (metamodels)
Conventional Wisdom

• the conventional wisdoms is that the two are distinct
  – ontologies are for some things and models for others
  – you need to make a choice

• there is growing interest in how ontology-based technologies and model-based technologies overlap
  – growing number of related workshops such as SEKE, VORTE, MDSW, SWESI, ONTOSE, WoMM, ...

• numerous proposals for “bridging” or integrating the two technology spaces
  – Ontology-driven Architectures
  – Ontology-based software engineering
  – Ontology Definition Metamodel
  – Model Driven Semantic Web

• all based on the premise that they are inherently different and there are objective distinction criteria
Common Informal Distinctions

- **Purpose-based distinctions**
  - models focus on realization (ontologies do not)
    - A Guide to Creating your First Ontology
      [Noy and McGuinness 2001]
    - ontologies are for run-time knowledge exploitation (Models are not)
      - Ontology Driven Architecture [W3C06]
  - ontologies are for representing shared (e.g. web based information) (Models are not)

- **Property-based Distinctions**
  - ontologies are formal (models are not)
  - ontologies can support reasoning (models can not)
  - models use the Close World Assumption (ontologies use the Open World Assumption)
A Guide to Creating your First Ontology (1/2)

• Highly influential paper by Noy and McGuinness describing how to create a Wine Ontology

• Step 1: Determine the domain and scope of the ontology
• Step 2: Consider reusing existing ontologies
• Step 3: Enumerate important terms in the ontology
• Step 4: Define the classes and the class hierarchy
• Step 5: Define the properties of classes - slots
• Step 6: Define the facets of the slots
• Step 7: Create instances
A Guide to Creating your First Ontology (2/2)

- although the authors acknowledge that some of the ontology design ideas originated from object-oriented design (Booch and Rumbaugh) they make the following distinction:
  
  **Ontology development**
  - reflects structure of the world
  - about structure of concepts
  - actual physical representation is not an issue
  
  **Object-oriented programming**
  - reflects structure of data and code
  - is usually about behavior (methods)
  - describes physical representation of data (long int, char, etc.)

- the described process is 100% the same as that for creating a domain (conceptual) class diagram
- the stored information is 100% the same as that which would be stored in a domain class diagram
Domain/Conceptual Modeling

• “A model of a system is a description or specification of that system and its environment for some certain purpose” (MDA Guide)

• "A domain model can be thought of as the conceptual model of the system……The domain model is created to understand the key concept of the system and to familiarize with the vocabulary of the system ” (Wikipedia)

• A domain model is an explicit description of a domain in terms of:
  – concepts
  – properties and attributes of concepts
  – constraints on properties and attributes
  – individuals

• A domain model defines
  – a common vocabulary
  – a shared understanding
Ontology Driven Architectures

- Quote from “Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering” by the W3C
The Role of OCL

- an often quoted shortcoming of the UML for the purpose of ontology representation is the lack of global properties
  - “it is not possible to show that different incarnations of a relationship (e.g. “owns”) are somehow the same”

- but although association inheritance is not directly supported, it can be simulated with the help of OCL

```
context Company inv owns1: owns->forAll(o | ooclIsTypeOf(Car))
context Landlord inv owns2: owns->forAll(o | ooclIsTypeOf(House))
```
Summary of Current Situation

• no consensus on the relationship between “models” or “modeling” and “ontologies” or “ontology engineering
• old distinctions based on “purpose” becoming obsolete
  – people shift criteria to suite their purpose and artificially maintain the separation
• most uses of UML as a visual notation for ontologies override (ignore) UML’s default semantics
  – metaphor overload
• why it matters
  – general confusion, arbitrariness and ubiquitous use of semantics-free statements
  – very early choice between OW and CW technologies
  – developers locked to one technology due to migration barrier
  – unnecessary weaknesses in both technologies
  – second order notions (OWL-S, CCM, UML Components ..)
Fundamental Definitions

Models

DM1. “A model is a homomorphic (or isomorphic) mapping of a subject matter into a system of symbols.”

DM2. “A model of a system is a description or specification of that system and its environment for some certain purpose”

Herbert Stachowiak

MDA Guide
Fundamental Definitions

Ontologies

DO1. "An Ontology is an explicit specification of a conceptualization."

Gruber et. al.

DO2. "An ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it IS not private to some individual, but accepted by a group."

Studer et. al.

DO3. "[...] an ontology refers to an engineering artefact, constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary. Usually a form of first-order-logic theory is used to represent these assumptions, vocabulary appears as unary and binary predicates, called concepts and relations, respectively."

Maedche
Observations

O1. Any Information representation (IR) that fulfils the conditions DO1, DO2 and DO3 for being an ontology also fulfils the requirements DM1 and DM2 for being a model. All ontologies are models, but not all models are ontologies.

O2. Most of the purpose-oriented characteristics are not mentioned in the core definitions. Support for reasoning and intelligent databases are not therefore required for conformance to DO1, DO2 and DO3.

O3. Of the informal property-oriented distinctions only formality is mentioned in the core definition DO3. No requirement for OWA or CWA for either models or ontologies.

O4. There is no mention of the intended scope of a model in either of the core definitions DM1 and DM2 (i.e. there is no reference to whether they are intended for representing shared information). However, there is no requirement that models are restricted to the representation of private (i.e. non shared) data.

O5. When supported by OCL, IRs in the MOF/UML can be created that satisfy DO1, DO2 and DO3 and thus can be considered ontologies based on the core definitions.

O6. IRs in OWL that have no shared understanding contradict DO2. It is therefore possible to create IRs in OWL that are not ontologies.
Ontology Set Membership Criteria

• the set of ontologies should be viewed as a subset of the set of models
  – not particularly controversial

• BUT, when does a model qualify as an ontology?
  – what is the ontology set membership criteria?
  – when should one refer to a model as an ontology?

• possessing the following properties is necessary, but is it sufficient?
  – conceptualization, explicit, machine readable, based on first order logic, shared

• if we say yes, then “domain models” or “computation independent models” qualify as ontologies
Proposal

- the term ontology be saved for models which satisfy these five criteria and are intended to be of “universal” scope
  - machine readable form of standard reference books

- Qualifying models
  - general ontologies
    - Mereology Ontology
  - upper-level” or “top-level ontologies
    - Suggested Upper Merged Ontology

- Non-qualifying models
  - limited scope models (e.g. SE domain models)
  - informal models
  - many taxonomies

- Metamodels are common examples of ontologies?
Choices Today

- When to use an “ontology technology” (Protégé, OWL)
  - if you need run-time support for automated reasoning (classification and subset recognition)
  - if you want to share your information on the web
- When to use a “modeling technology” (UML Tool, UML)
  - if you want an intuitive and expressive graphical notation (but don’t need reasoning support beyond inheritance)
  - if you want to construct artifacts using MDA techniques (including database tables)
- if you want UML / MDA features and support for reasoning you have a problem
- in general, these choices are overridden by more pragmatic concerns
  - employee experience/training
  - upwardly compatibility
  - marketing
Vision (or Hope)

- unified information representation technology which supports
  - same graphical syntax for common object-oriented representation concepts (classes, relationships ..)
  - most general default assumptions
  - ability to switch between OW/CW assumptions
    - expressed as a general constraint at the appropriate meta level
    - maximum possible degree of reasoning

- consensus on when to label an IR as an ontology and as one of the different kinds of model
  - Domain/conceptual models versus ontologies?
  - move away from current arbitrariness and semantics-free use of terminology
End of lecture 4