Best Practices for Applying UML, Part I

Darius Šilingas, Ph.D.
Principal Trainer for MagicDraw UML

magicdraw™
Architecture Made Simple

Content

Although UML notation is widely recognized as *lingua franca* for software development, many developers still lack skills for applying it efficiently. Methodologists, practitioners and tool vendors need to share their experience in order to find out ways how to get the most use from modeling and avoid abusing it. With this whitepaper, we start a series of best practices in applying UML that are driven from No Magic expertise in multiple software development projects and building MagicDraw UML tool. We hope that they will help you to change the way you model and make it more efficient!

Software Development and Modeling with UML

Modeling is the core of modern software development. Like every larger engineering task, software development needs a plan. Models serve us as a plan and allow achieving multiple goals:
- Provide structure for problem solving;
- Experiment to explore multiple solutions;
- Furnish abstractions to manage complexity;
- Reduce time-to-market for business problem solutions;
- Decrease development costs;
- Manage the risk of mistakes.

Only when developing very small systems, we may start from scratch and bypass modeling.

You **DON'T** need modeling	You **DO** need modeling
Small one-person projects that do not need to be maintained and will not be transferred to somebody else might not require modeling.	Large projects that involve large teams, are developed iteratively, and need to be maintained cannot be successful without models.

In large projects modeling speeds up implementation while elevating quality and user satisfaction. We model in all software development activities starting from business analysis and requirements definition to testing of the constructed software system.
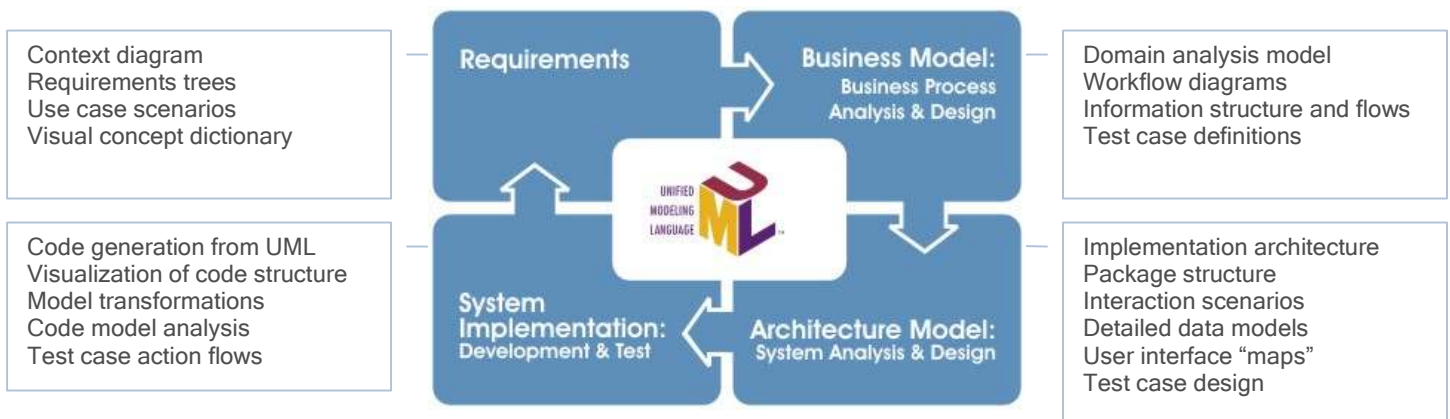

Context diagram
Requirements trees
Use case scenarios
Visual concept dictionary

Domain analysis model
Workflow diagrams
Information structure and flows
Test case definitions

Code generation from UML
Visualization of code structure
Model transformations
Code model analysis
Test case action flows

Implementation architecture
Package structure
Interaction scenarios
Detailed data models
User interface "maps"
Test case design

Requirements

Business Model: Business Process Analysis & Design

UNIFIED MODELING LANGUAGE UML

System Implementation: Development & Test

Architecture Model: System Analysis & Design

Table 1– Applicability of UML Modeling Skills

Software systems are constantly growing in size and complexity, and without efficient modeling skills you cannot be successful. A Software Development Magazine Salary Survey in 2005 has revealed that the most important skills needed to produce quality projects and products are architecture modeling and design, which ranked higher than programming, QA, testing, and requirements management.

Recently, we witness intense battles in the field of the software development process. While almost all approaches have gone from the old waterfall approach to iterative development, they argue a lot about formality, roles, activities and artifacts, tools, and which workflows should be emphasized.
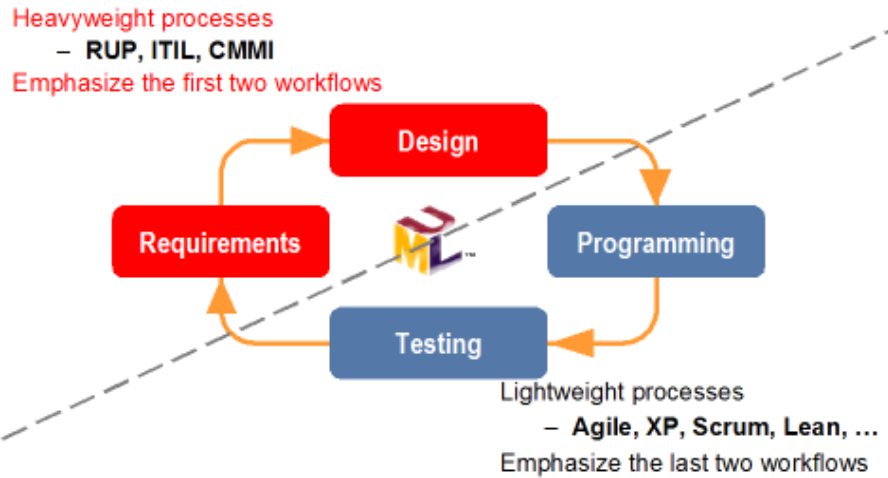
Figure 1 – Emphasis on Workflows by Different Softwate Development Processes

Since modeling is the core of development and UML is the *de facto* standard as the modeling notation, all of those processes make use of UML. However, the way they use it differs quite a lot. RUP advocates a UML centric approach to software design, which requires UML modeling tools. Agile methods – XP, Scrum, Lean – advocate informal modeling and using whiteboards, paper, and simple text or drawing editors as a compliment to UML tools. While we are in none of these camps, we strongly believe that using modeling tool can help you a lot.

Here is just a short list of helpful UML tool features:

- Create and maintain a repository of model data;
- Reuse model elements in multiple diagrams;
- Create nice diagrams using automated layout functions;
- Apply built in design patterns;
- Use automated model transformations;
- Generate documentation from your model;
- Analyze your model using automated tools;
- Enable teamwork within one modeling project.

We believe these features are truly beneficial in all software development processes if you take a proactive approach to modeling. In the next section, we present 5 best practices for modeling with UML.

Best Practices

Experience is the worst teacher – it always gives the test first and the instructions afterward.

Let's get some instructions before you apply UML in your projects – you can count on our experience, which resulted in the best practices presented below.

Best Practice #1: Apply a subset of UML relevant to your role

80%–20% rule: 80% of the users apply only 20% of the features.

UML 2.0 defines 13 diagram types and each of them uses a bunch of UML elements. You can find many people arguing that UML has grown too large and too complex. Well, it definitely evolved and contains more modeling elements if compared to its original version. However, it is important to recognize that you don't in most cases need the full capabilities of UML. The famous 80%–20% rule holds here as well – 80% of modelers will be fine with 20% of UML. Which 20% is yours is dictated by your role and that role's need for specific UML modeling skills.

Table 2 – Applicability of UML Modeling Skills

Business Analysts	Software Architects
Capture business processes in diagrams Model organizational structures Perform analysis of domain concepts Prepare use case-driven requirements model ...	Relate different architectural views Transition from business to system models Define top-level components and package structures Manage modeling teamwork …
Developers	**Quality Assurance Engineers**
Prepare detailed class models Model interactions for use cases Introspect existing systems by reversing and visualizing code Design OO, relational, and XML data Transform class structures to database or XML schemas …	Analyze workflows for use cases Prepare action flows for test cases Model test data Model interactions for unit and system testing …

Business analysts will mostly apply *use case* and *activity* diagrams for capturing processes and requirements, also very simple *class* diagrams for domain analysis and *state* diagrams for defining lifecycle of important business objects. They will need extensions of UML for modeling enterprise architecture concepts like *organizational structure* or *business goals*.

Software architects will apply all UML diagrams, but they would not go into implementation details. Architects need to capture overall system structure in high level concepts. In most cases, they will be fine with *package*, *component*, and *deployment* diagrams; and will model only the most important *classes* and *interactions*. When building classical layered systems, they might use the *robustness* diagram – a simple extension of the UML class diagram – for identifying major components in interfaces, business logic, and data layers.

Developers will not model all parts of the system, but they would go into low level implementation details on the most important system parts, e.g. data structures. They use detailed *class* and *interaction* diagrams. Most probably, they will also need UML extensions for capturing technology specific details like relational database structures or XML schema elements.

Quality assurance engineers need to analyze UML diagrams prepared by business and system analysts, and base their test cases on *use case* workflows. They will need *object* diagrams for modeling test data and *activity* diagrams for modeling test action flows and rules. For those who do more technical testing like unit testing (though in the most of recent approaches it is done by programmers), stress or load testing might need to model interactions using either *sequence* or *activity* diagrams.

The lesson learned

You don't need to know and use the complete UML specification. Learn and apply only the subset that is relevant to your role!.

Best Practice #2: Focus on the Most Valuable Modeling Artifacts

The cynic knows the price of everything and the value of nothing. [Oscar Wilde]

The second best practice supplements the first one. Knowing what gives value to your role, you also need to know what modeling artifacts have long-term value and need to be maintained. The others might be used for instant needs and thrown away later. Understanding this is really important, since trying to keep too many irrelevant modeling artifacts synchronized with actual source code quickly becomes *a pain in the … archive*. We believe this synchronization headache makes many people think they don't need modeling tools and are fine with whiteboards. But they soon get lost in whiteboard discontinuities.

However, analysis model artifacts are of high value and should be consistently modeled and maintained, including analysis artifacts such as:

- Domain entity relationship (ER) class diagrams;
- Use case models;
- Activity diagrams for use case scenarios.

Business analysts perform domain analysis by modeling domain entities and their relationships using simplified *class diagrams*. These diagrams serve as a visual vocabulary of concepts. They are also a starting point for the design level data model.

MagicDraw tips

MagicDraw UML tool can even generate a dictionary from class descriptions in HTML format if you provide concise descriptions for modeled concepts.
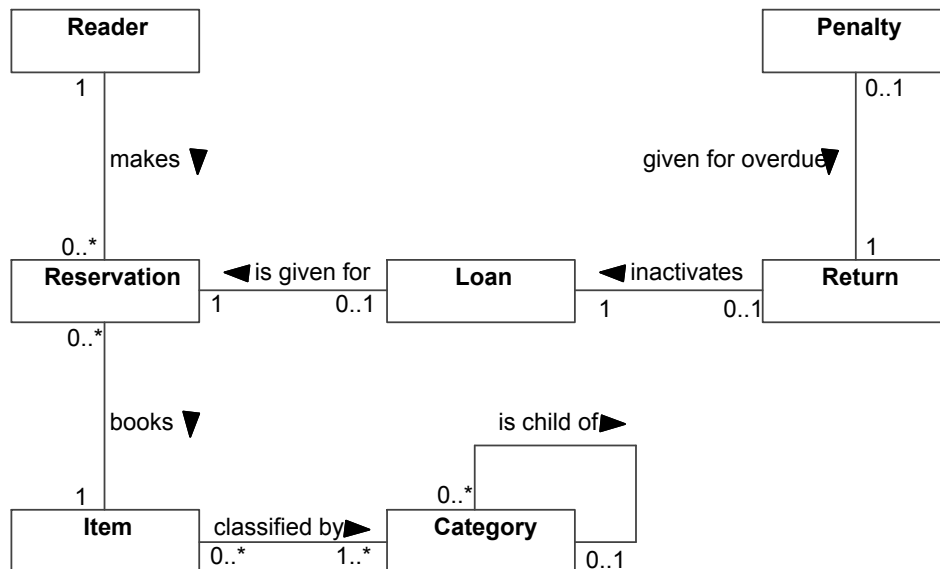

Figure 2 – Domain ER diagram for library management system

The most popular way to analyze functional requirements is the *use case method*. It starts by identifying user roles – actors – and associating them to their interactions with system – use cases. Many business analysts even start with Use Cases as the best way to capture and document user requirements.

Abstract use case implementation logic is either defined in text or, for more complex cases, by creating activity diagrams visualizing activity flows for primary and secondary use case scenarios.
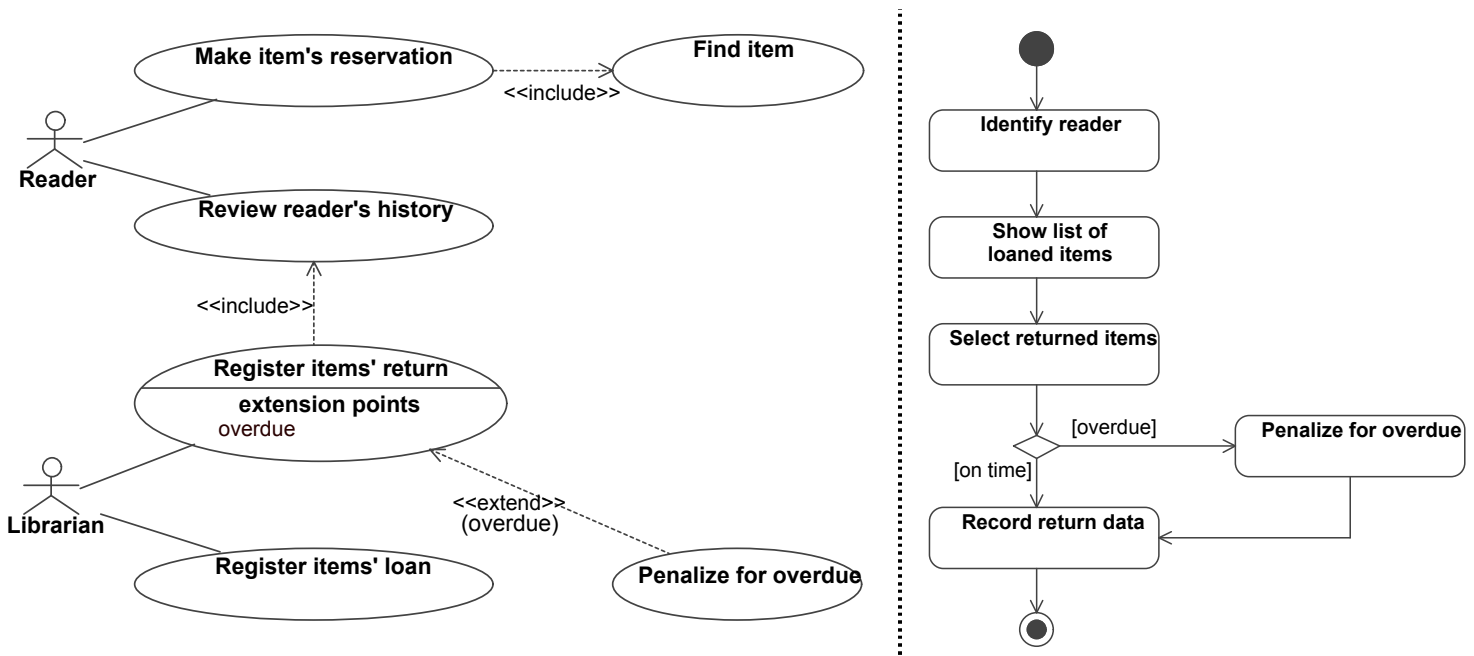
Figure 3 – Use case model artifacts: *use case diagram* and *activity diagram* presenting scenarios for a particular use case *Register items return*

Technical design model artifacts may be roughly divided into three value-based groups:

- *High value* ($$$) – should be modeled and maintained using UML tools;
- *Medium value* ($$) – may be maintained or not;
- *Low value* ($) – should be used for communication and thrown away.

$$$ – High value design models

- Top level structural decomposition of software and hardware environments
    - UML package, component, robustness and deployment diagrams
- Top level package dependency diagrams
    - UML package diagram
- Data model
    - UML class diagrams in multiple level of abstraction and details
- Interfaces for system layers / components / subsystems
    - UML class and component diagrams
- Dynamic models for interactions implementing main usage scenarios
    - UML activity and sequence diagrams (details, e.g. parameter values, are often left out)

System architects need to bridge the gap between analysis and design. If they are designing layered systems, they may apply *robustness analysis*, which is based on a simple extension of the class diagram. It makes use of only four elements – *actors*, *boundaries* (user or system interfaces), *controls* (business logic), and *entities* (data). While entities are more or less identified by the domain ER analysis, the challenge is to analyze actors and use cases, find the system boundaries, and connect them to data entities through controls.
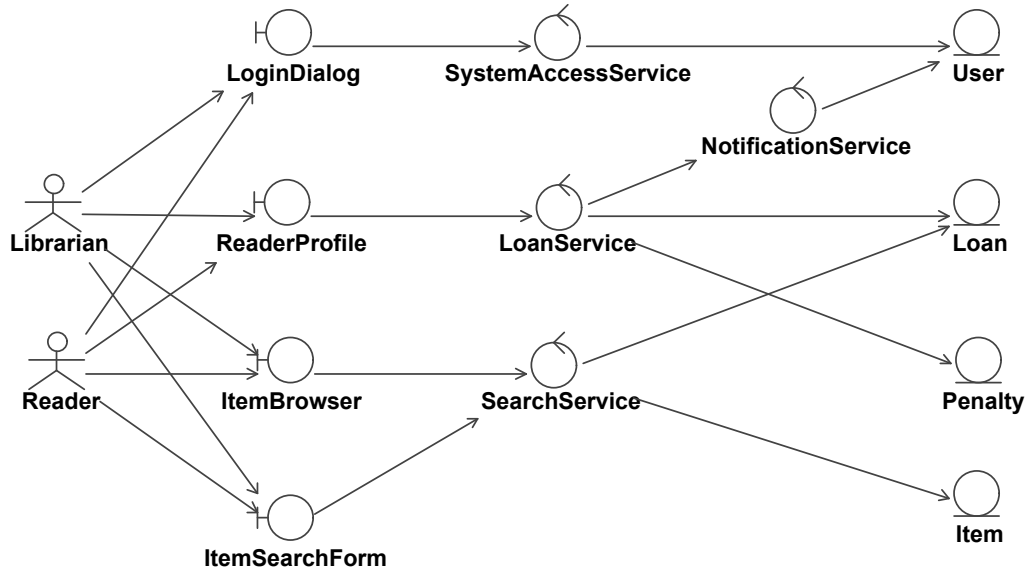
Seeing the forest before going inside it.

Figure 4 – Robustness analysis for identifying major components in a layered architecture

Architects need to provide the top-level decomposition of software system packages. Modeling dependencies are very important in this case since they set the rules for class responsibilities and interactions and help avoid evil effects like inappropriate coupling and cyclic dependencies. The tools for showing decomposition including dependencies are the *package diagram* and the *component diagram*.

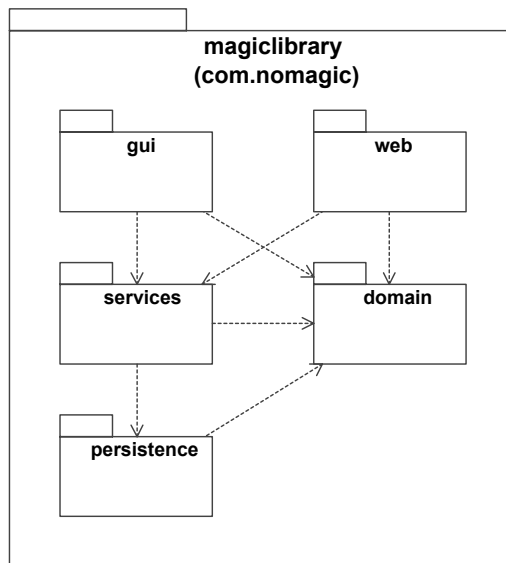*Divide and rule! - a strategy used by ancient Roman empire.*

Figure 5 – Package dependency diagram

While the package diagram allows a logical grouping of implementation classes, the component diagram shows system level functional elements. Usually, package and component diagrams leave out utility or library packages. Although packages should be component driven, one package may include multiple components and vice versa. For example, a component might span several packages depending on its size. Component diagrams may also focus on the provided and required components, and on connecting the components together.

It is also important to show how implemented software will be deployed in the hardware and software environment. For this purpose, architects apply *deployment diagrams*. In UML 1.x, deployment diagrams allowed nesting components on deployment nodes. In UML 2.0, this has been changed, and instead of components, you need to use artifacts that are a physical manifestation of software components. It is a rather common practice to create technology specific UML extension profiles. Profiles enable one to indicate a 'type' of an artifact, e.g. <<*EJB-JAR*>>, and preferably to assign an intuitive icon, which can replace the standard rectangle shape, e.g. a <<*database*>> icon.
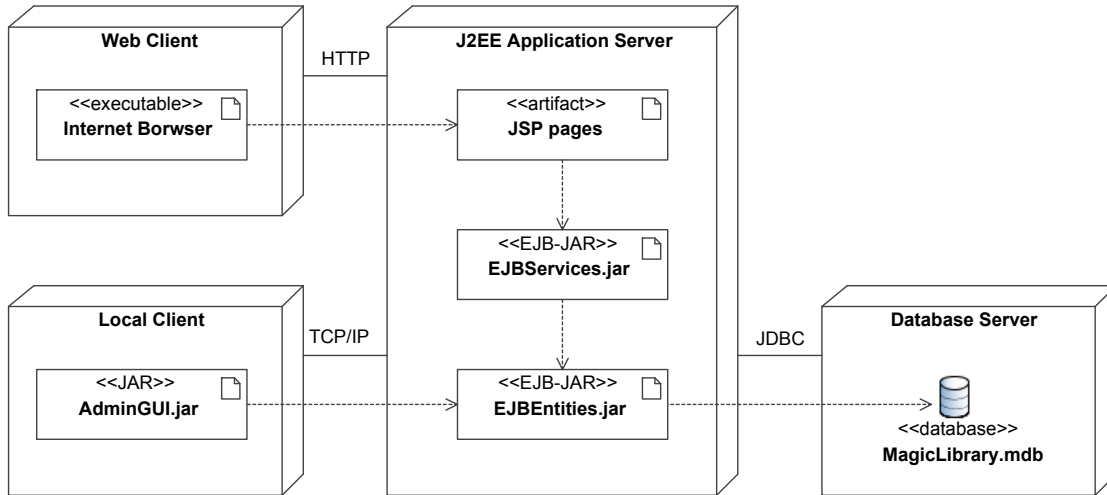
Figure 6 – Top level deployment diagram

Once architects have decomposed the system into packages, components, and deployment artifacts following the layered architecture principle, the next important thing is to specify the contracts between layers. The most important piece here is the method specifications for business logic services, and also for data persistence services if they are not reused. The first step can be done using *class diagrams*.
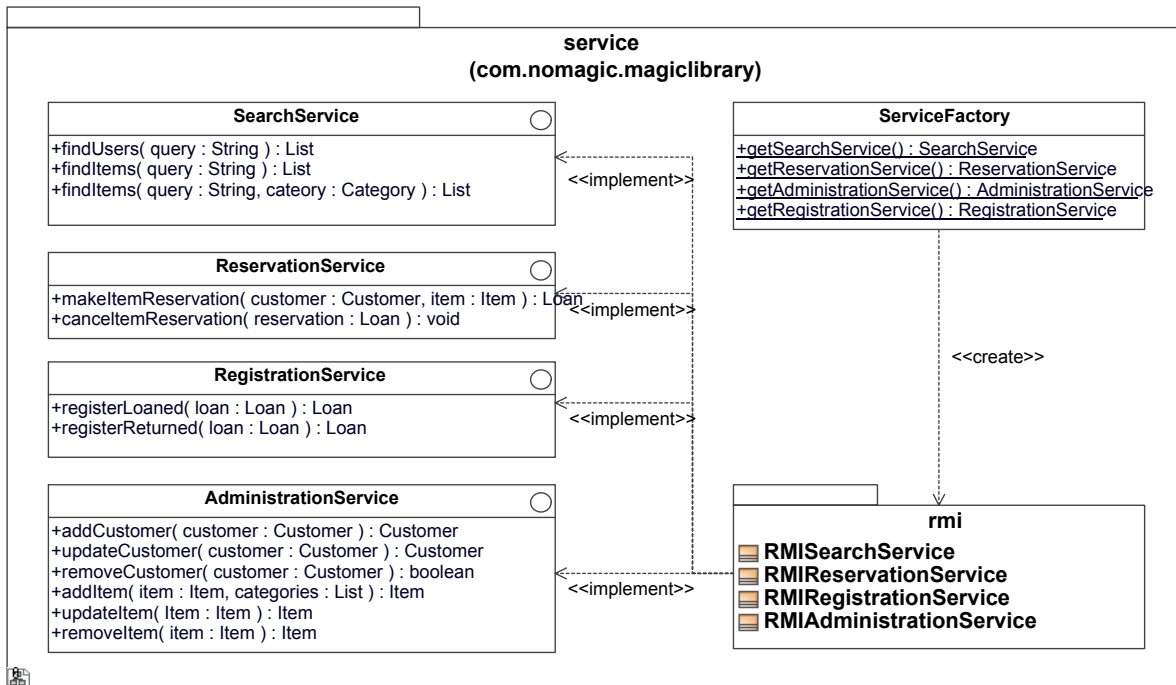
Figure 7 – Interface specifications for business logic layer components

However, using only static class diagrams is not sufficient – you may miss some responsibilities or decompose them inappropriately. Therefore you need to model the dynamic behavior for use case scenarios in order to validate and justify class structures. Applying interaction diagrams – *sequence* or *communication* – help in find new methods in existing classes, and in finding new classes for specific responsibilities. Interaction driven modeling is actually a very efficient method, and is the modeling counterpart for test driven programming.

MagicDraw Tips

MagicDraw allows creating new classifiers and adding new methods to existing ones when modeling interactions. You can also easily drag and drop classes to create typed lifelines.

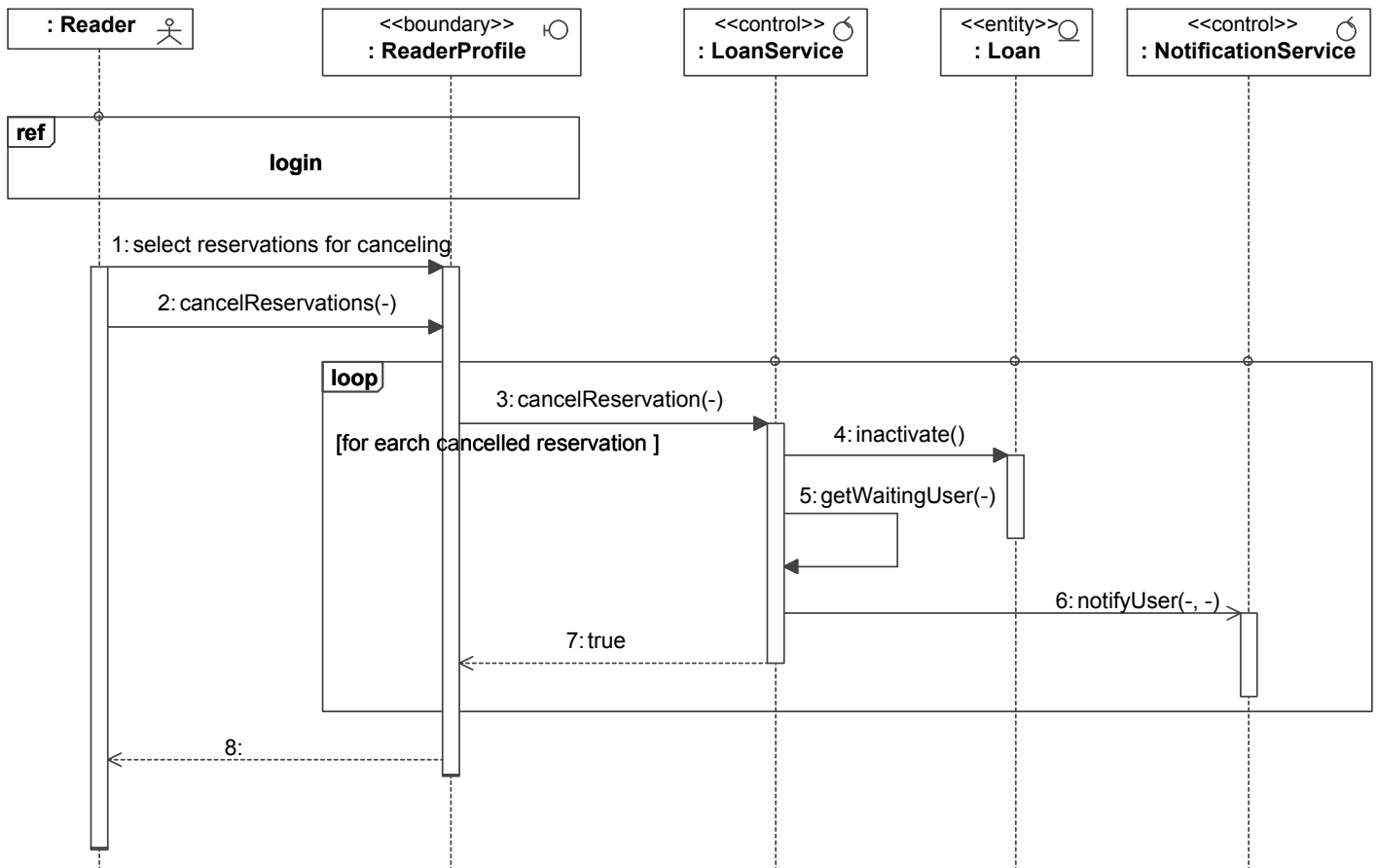**Figure 8 – Interaction diagram for implementing specific use case scenario**

$$ – Medium value design models

- Structure of UI classes
  - UML class diagram, might be extended for specific technology
- UI screen transition schema
  - UML state or class diagram
- Reusable implementation solutions
  - UML class, object, sequence diagrams
- Logic of important algorithms
  - UML activity or sequence diagrams

During technical design activities, if you go up the layers of the software system, the stability decreases. Detailed models for user interface layers are volatile during design phase – they change a lot, and throwaway prototyping is an approach used. Whether this method is used or not, is it still valuable to model high-level structures of user interface elements without going into details.
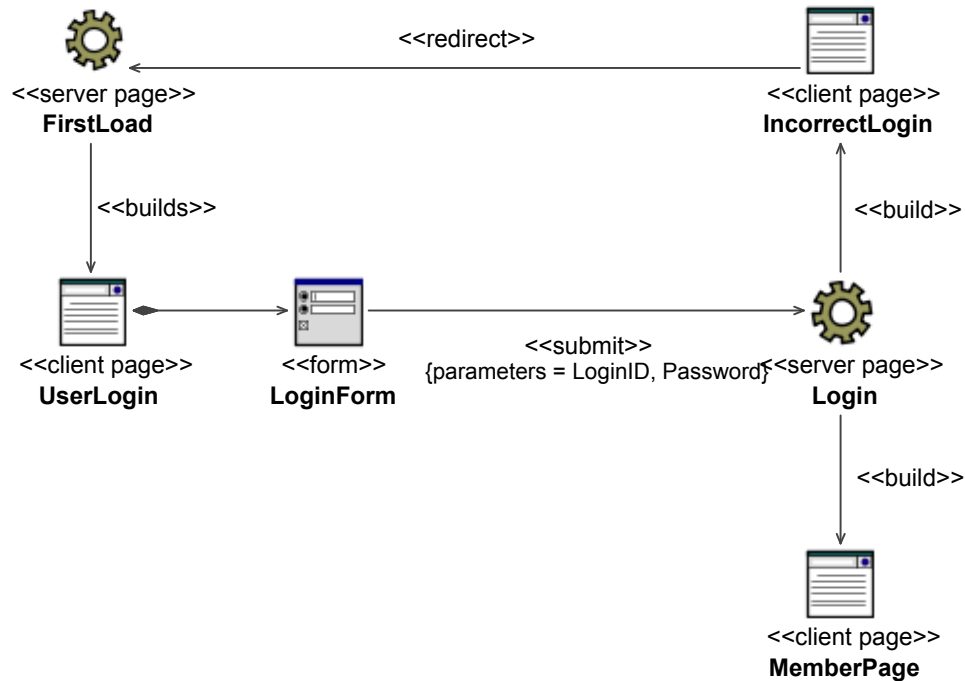
Figure 9 – Structure of Web UI Classes

It is rather difficult to get precise requirements for the system without prototyping its user interface and sketching its navigation schemas. Thus, drawing and maintaining a navigation schema for a system is also a highly useful part of your modeling toolkit. You should apply *state diagrams* for this task.

Figure 10 – User Interface Dialog Map

If you go deeper into implementation, modeling is also essential, and it can also be accomplished using UML tools. For example, finding the methods for implementation classes can be accomplished using test-driven programming, which forces you to write unit tests and only then implement the class. Having a model of the tests and test cases helps navigate

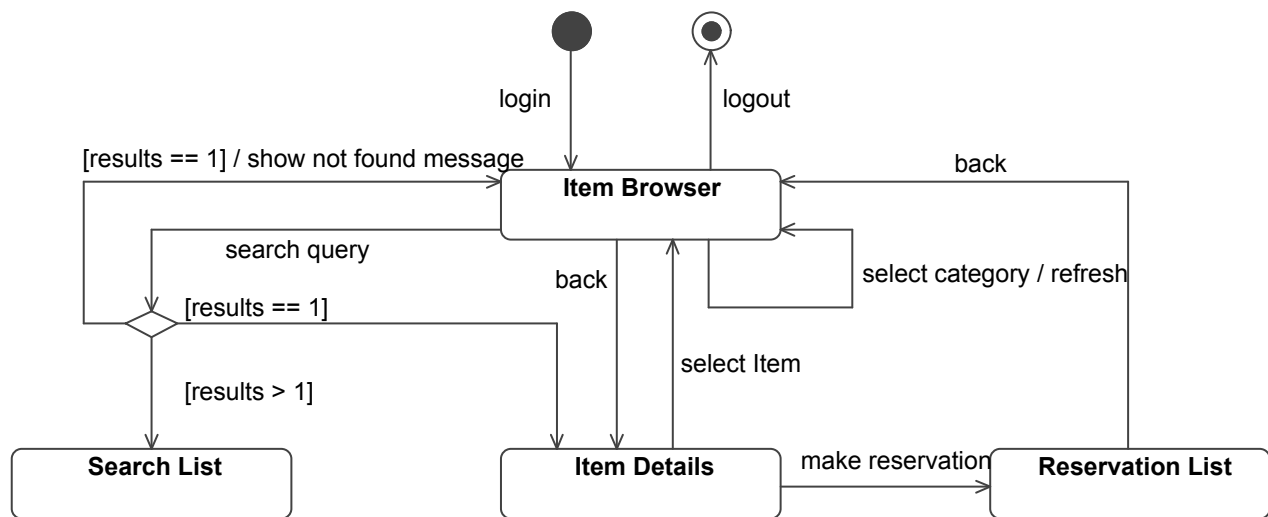through the test process. The testing functionality has excellent support in modern IDE tools. However, UML models illustrating the logic of important algorithms or reusable implementation solutions may also be very helpful for various purposes – analysis, learning, and documentation.

> **$ – Low value design models**
>
> - Detailed implementation level class modelsq
> - Dependency diagrams for "deep" inner packages and classes
> - Detailed user interface models
> - Interaction diagrams that model all details of implementation
> - Trivial activity or state diagrams
> - …

During technical design phase, detailed models change frequently, thus it is better to model, implement, and analyze them in conjunction with programming IDE tools. Most of the technical models can be reversed and visualized automatically from source code using code reverse engineering tools.

MagicDraw Tips

MagicDraw UML supports forward and reverse engineering of various types of code: Java, C++, C#, relational databases, XML schemas. It also provides multiple model visualization tools that in an automated way create multiple diagram types such as package dependency, class hierarchy, and Java sequence diagrams.

The lesson learned

Model artifacts may be difficult to maintain, thus you need to focus on the most valuable ones. Draw and maintain the $$$ value artifacts using UML tools, make your selection of $$ value artifacts that you find useful in your projects, and don't use for sketching but don't worry about maintaining $ value artifacts. Maximize your time and your efforts in this way.

Best Practice #3: Model in Multiple Abstraction Levels

Abstraction is the process of reducing the information content of a concept, typically in order to retain only information that is relevant for a particular purpose [Wikipedia]

Abstraction is the thought process wherein ideas are distanced from objects [Wikipedia]

We model to furnish abstractions in order to manage complexities. Usually there are several levels of abstractions that need to be modeled. The following abstraction levels are usually defined:

- *Analysis Model* aka *Computation Independent Model* (CIM);
- *Design Model* aka *Platform Independent Model* (PIM);
- *Implementation Model* aka *Platform Specific Model* (PSM).

In your UML modeling tool, you should structure your model so that these abstraction levels were clearly separated, easily maintained, and backtracked. Let us show an example of traveling through these three abstraction levels.

The conceptual library management system ER diagram is an example of the Analysis Model (CIM). It shows the concepts that exist in the library system independently of whether it is managed manually of with software system. The links show logical relationships, and we do not model attributes. Even if attributes were modeled, their data types would not be assigned.
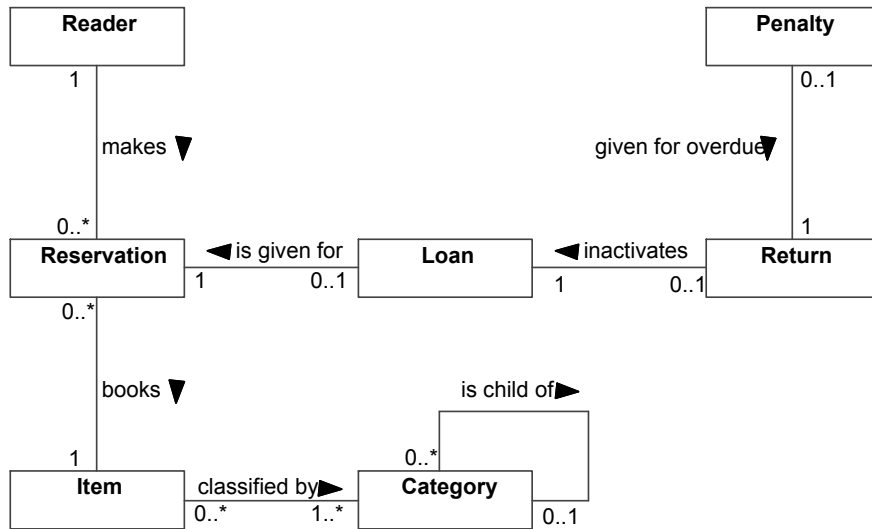
Reader 1

makes ▼

0..* **Reservation** 1 ◄ is given for **Loan** 0..1 1 ◄ inactivates **Return** 0..1 1

Penalty 0..1

given for overdue ▼

0..*

books ▼

1 **Item** classified by ▶ 0..* 1..* **Category** is child of ▶ 0..*

0..1

Figure 11 – Analysis model of library concepts (CIM)

When designing the software system, we need to derive a detailed data model. It has to be conformant with the analysis model, but some restructuring may be applied based on design decisions. For example, the diagram below illustrates a design model derived from the previous analysis model by making the following changes:

- Attributes with UML data types were specified for each class;
- Concepts of *Reservation*, *Loan*, and *Return* were merged into one class *Loan*;
- Two specific subclasses of *Item* – *Book* and *AudioRecord* – were added.

This model already defines the complete data structure, but is not technology specific – it's pure object oriented structure with UML data types, thus not specific to any particular programming language like Java or data storage technology, relational database, or XML.
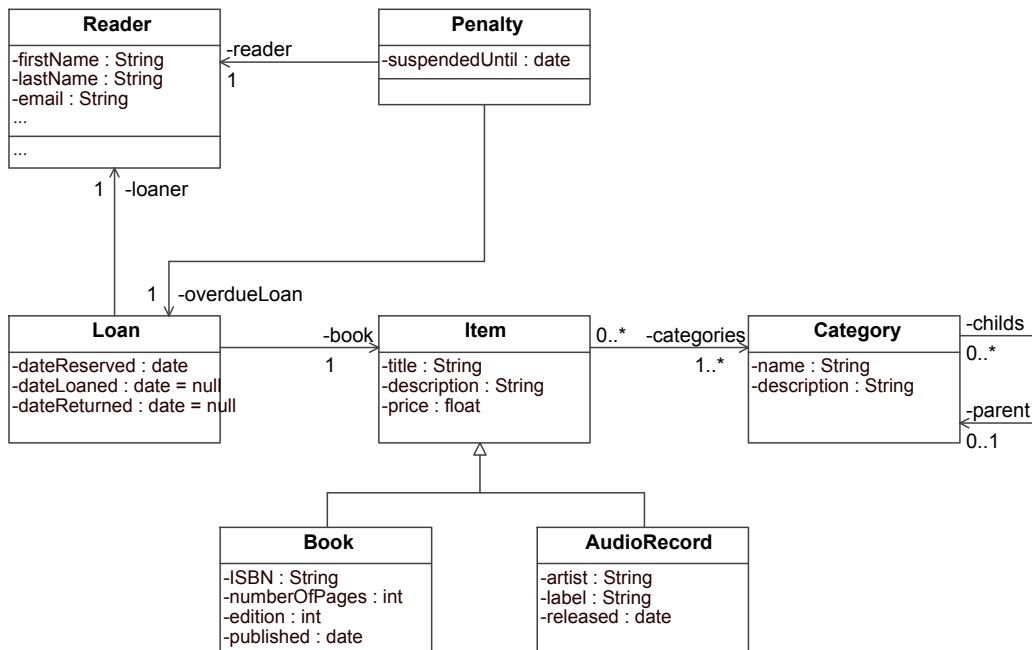
Reader
-firstName : String
-lastName : String
-email : String
...
...

-reader
1

Penalty
-suspendedUntil : date

1 -loaner

1 -overdueLoan

Loan
-dateReserved : date
-dateLoaned : date = null
-dateReturned : date = null

-book
1

Item
-title : String
-description : String
-price : float

0..* -categories
1..*

Category
-name : String
-description : String

-childs
0..*

-parent
0..1

Book
-ISBN : String
-numberOfPages : int
-edition : int
-published : date

AudioRecord
-artist : String
-label : String
-released : date

Figure 12 – Design model of a library data structure (PIM)

The next step is to derive the platform specific model, which contains technology specific details and has enough precision for generating executable code. For this kind of modeling, you need profiles with extensions to the UML

language, which are technology independent.   The example below shows a relational database structure, which corresponds to the design model, with the following changes:

- Classes were assigned stereotypes <<table>>;
- UML data types were changed to SQL data types and multiplicities were identified;
- Primary and foreign key fields were added to classes;
- The associations were assigned stereotype <<FK>> with properties indicated field mapping;
- The many-to-many association was changed into intermediate table;
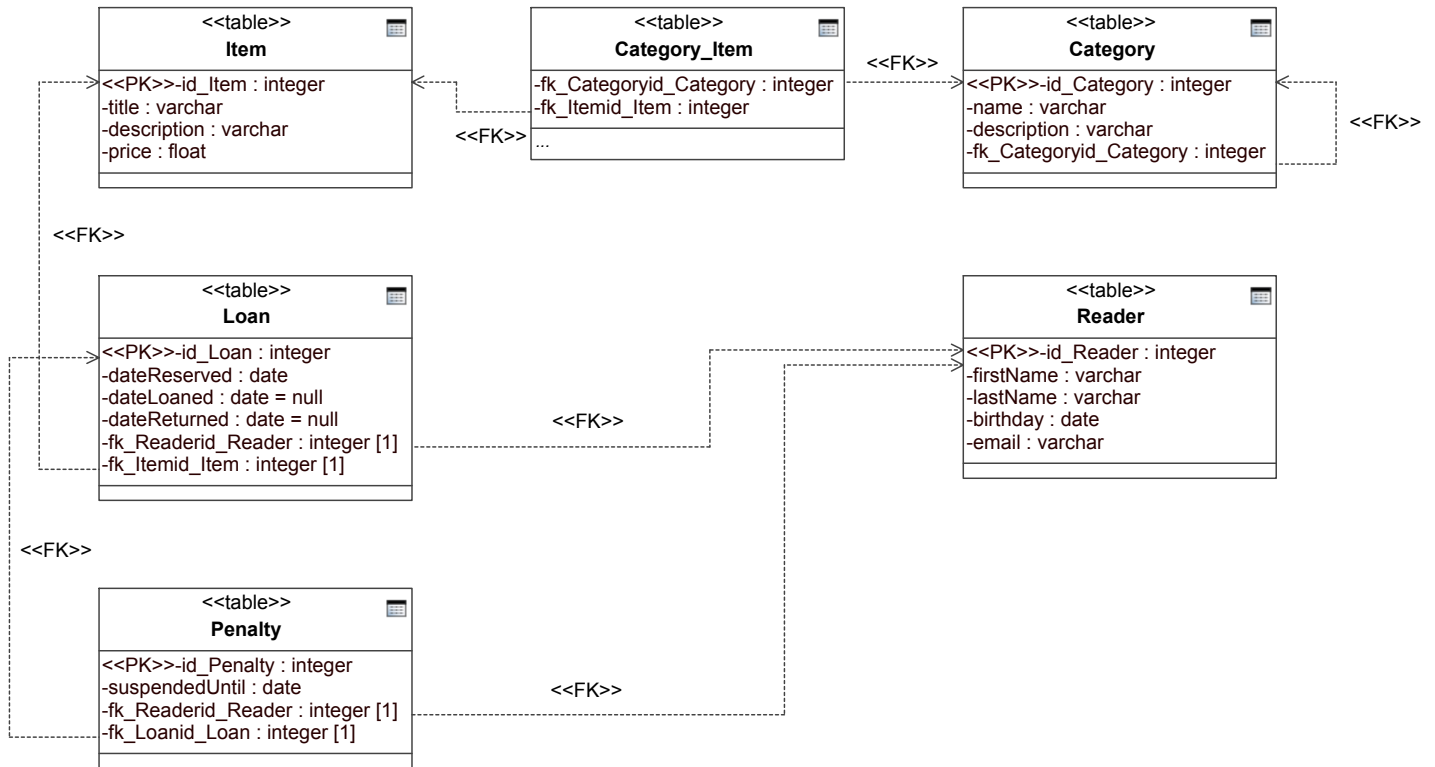- The generalization relationships were turned into foreign keys.

Figure 13 – Implementation model of library relational database schema (PSM)

Although it is possible to model the PSM manually, it is advised to use automated transformations. However, using transformations is a topic of another best practice in the upcoming whitepaper.

The lesson learned

You can model at different abstraction levels. However, it is crucial to decide how many abstraction levels you need for you project – some will be fine with one, but the most will at least require two if not three. Clearly separating abstraction levels in your model projects facilitate maintaining and backtracking related elements.

Best Practice #4: Choose Appropriate Level of Detail

Figure 14 – Diagrams overloaded with details are difficult to read and understand

Level of abstraction dictates the level of detail. However, with UML modeling tools even within one abstraction level you may choose to show different levels of details. The purpose of what you want to show should drive the choice. Irrelevant elements that can be guessed implicitly should be left out. A common technique is to show a diagram without details of elements but showing the most important relationships between them. Then you can have additional diagrams that focus on the details of specific element.

Let's go through an example of multiple levels of details so that you can decide, which level is the most appropriate for you.
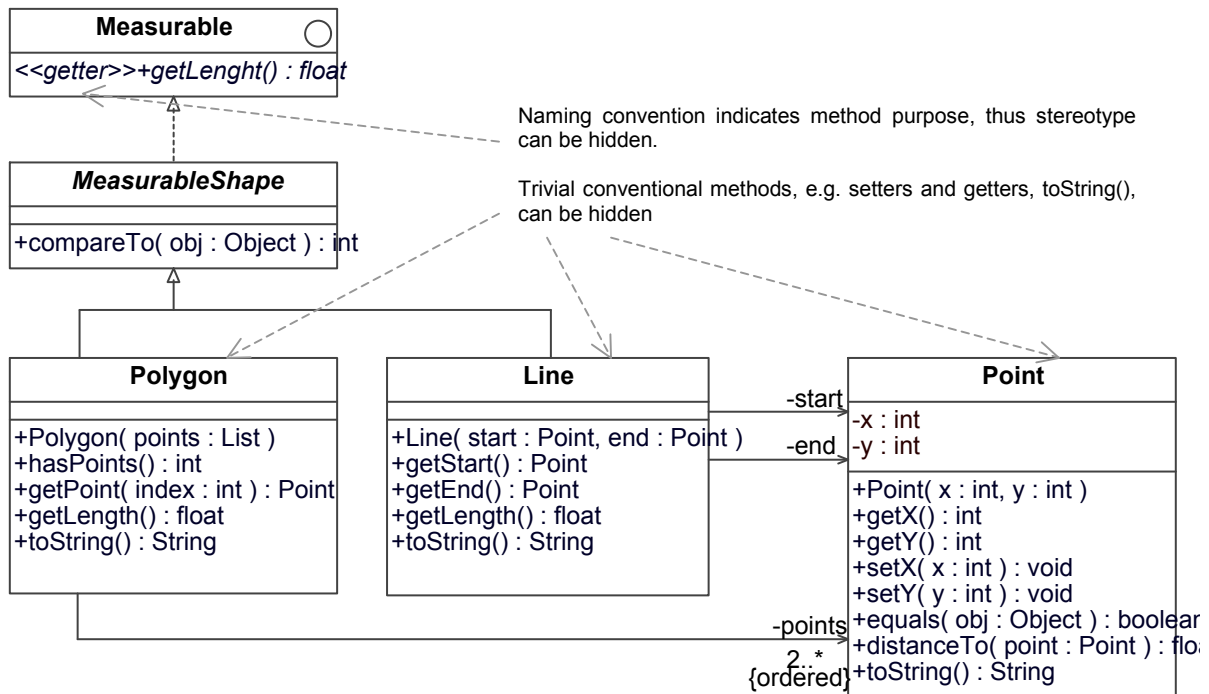
Step 1

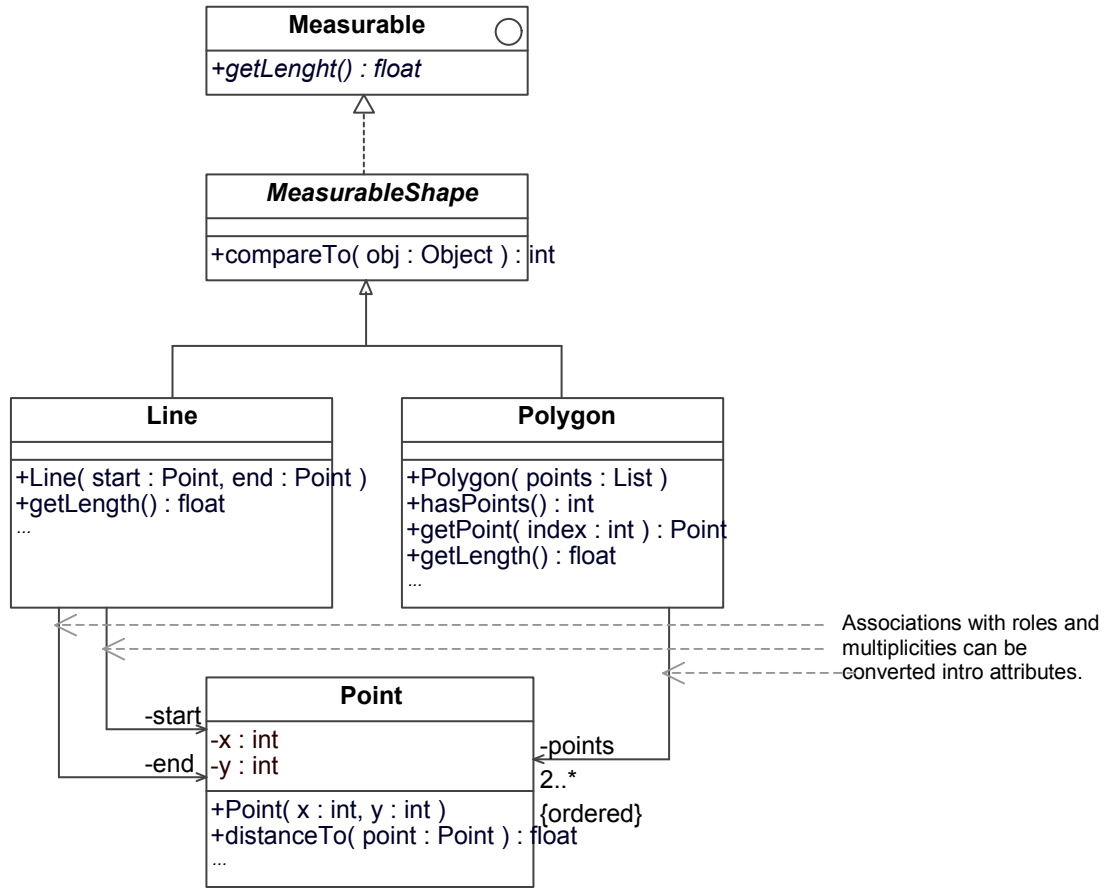**Figure 15 – Level of Details, Step 1: all implementation details are shown**

Step 2

Measurable

+*getLenght() : float*

MeasurableShape

+compareTo(obj : Object) : int

Line

+Line(start : Point, end : Point)
+getLength() : float
...

Polygon

+Polygon(points : List)
+hasPoints() : int
+getPoint(index : int) : Point
+getLength() : float
...

Associations with roles and multiplicities can be converted intro attributes.

-start

Point

-x : int
-y : int

+Point(x : int, y : int)
+distanceTo(point : Point) : float
...

-end

-points

2..*

{ordered}

Figure 16 – Level of detail, step 2: irrelevant elements are removed

Step 3

Measurable

+*getLenght() : float*

Associations with roles and multiplicities can be converted intro attributes.

MeasurableShape

+compareTo(obj : Object) : int

Line

-start : Point
-end : Point

+Line(start : Point, end : Point)
+getLength() : float
...

Polygon

-points : Point [2..*]{ordered}

+Polygon(points : List)
+hasPoints() : int
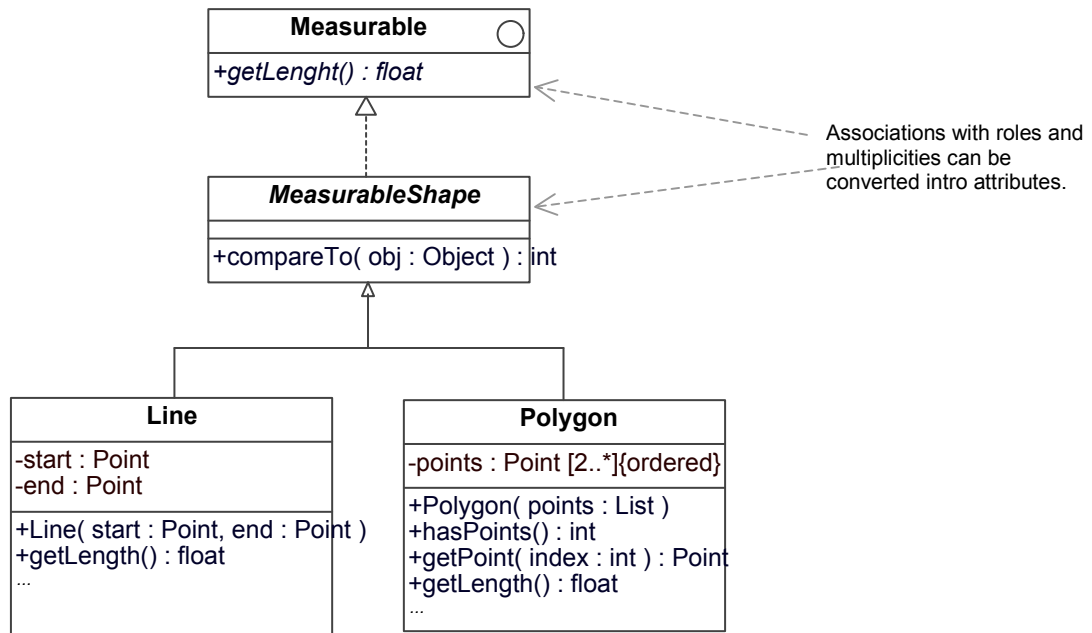+getPoint(index : int) : Point
+getLength() : float
...

Figure 17 – Level of detail, step 3: unimportant associations are turned into attributes
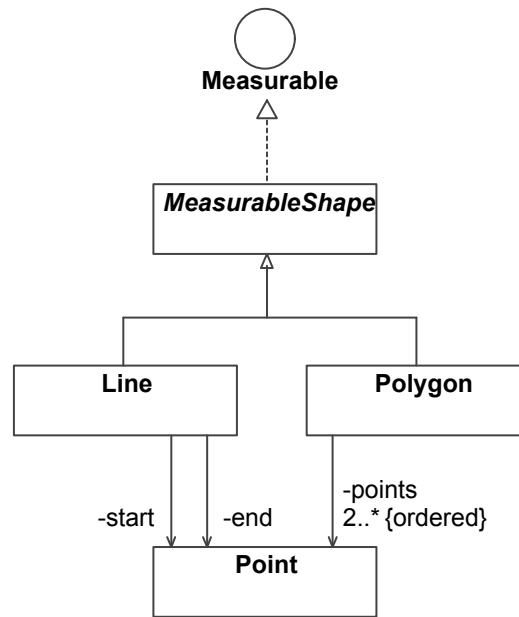
Step 4

Figure 18 – Level of details, step 4: only class and interface symbols and relationships between them are shown

As you probably agree, removing irrelevant details allows focusing on the essential design elements, which makes diagram easier to read and understand.

It is very important to choose appropriate level of details. We can give the following pieces of advise for controlling level of details in your diagrams:

- Do not model irrelevant details, such as:
    - Implementation-specific properties;
    - Private class methods and attributes;
    - Conventional methods, e.g. *toString()* in Java;
    - Utility classes;
    - Elements that can be generated automatically;
    - Getter and Setter methods;
- Show only the most important associations.

Too detailed models are difficult to understand and keep in sync. Therefore they are rarely used. As a result, it is common to give up their maintenance, thus they become worthless since they do not correspond to the actual design or code.

The lesson learned

Show only important properties of the model elements in your diagrams. Getting rid of irrelevant details improves communication and makes maintenance easier.

Best Practice #5: Model with Style

Style is the way you put together things, not the things themselves.

Software developers follow defined styles for programming, requirements specification, technical documentation, and user manuals. We also need modeling style as well. Adhering to common modeling style makes diagrams easier to read and understand and facilitates teamwork in modeling. Modeling styles includes:

- Naming conventions;
- Layout conventions;
- Choosing appropriate diagrams;
- Modeling & design principles.

A seminal book published on this topic is "*The Elements of UML Modeling Style*" by Scott W. Ambler. It gives a bunch of rules that need to be followed when modeling UML diagrams. We will present only an example of improving modeling style and identify the functionality of UML modeling tools that facilitate layout of diagrams.
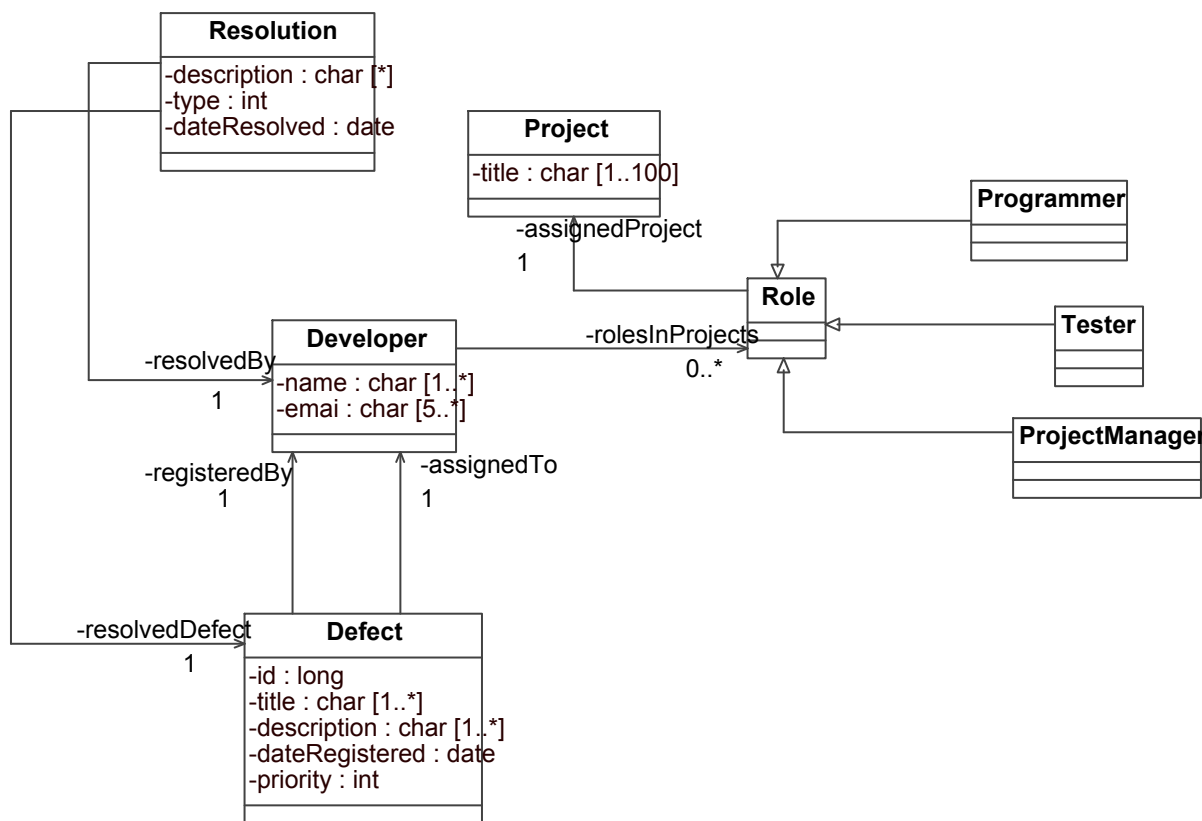

Figure 19 – A diagram which DOES NOT follow modeling style rules

The following problems are apparent in this diagram:

- Different element sizes are used;
- Diagram is not symmetric;
- Long, bent, and crossing lines;
- Horizontal ungrouped generalization links;
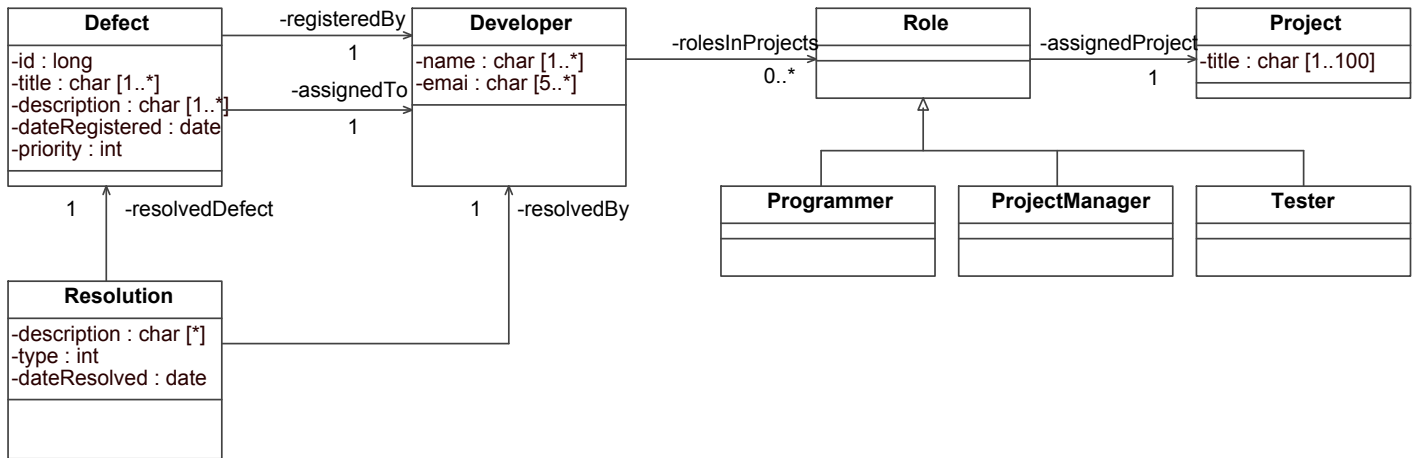- Unevenly distributed space.

Figure 20 – A diagram, which follows modeling style rules

Some sample modeling style rules:

- Do not overload diagrams with irrelevant details
- Split large diagrams into multiple smaller diagrams
    - Diagram should contain 7±2 elements
- Use colors to identify elements of different type
    - Color may indicate, which development group is responsible for implementation
- Try to maintain symmetry
- Model generalizations and realizations vertically
    - Child / implementing class should be below parent / interface
- Model associations and dependencies horizontally
    - Recommended direction – left to right

There are a lot more rules of thumb on modeling class and other diagrams – check Ambler's book or his website at http://www.agilemodeling.com/style/

For improving diagram layout, you may apply automated layout tools. However, relative positions of elements in most cases are very important, and most fully automated layout tools do not retain them. We have found out that simple layout functions applied to a selected area or selected elements in a diagram are most helpful:

- Make sub trees of generalizations, realizations, association;
- Make elements the same width, height;
- Distribute space evenly;
- Align elements to the top, bottom, left, and right;
- Route paths orthogonally;
- …

Sure, the content is more important than the form. However, sticking to one consistent modeling style makes the content of your diagrams more readable and understandable. Since modeling is both an interactive and iterative activity, diagram readability is essential for improving the content!

> ***The lesson learned***
>
> *Modeling with style improves the value of diagrams by increasing their readability. There is a lot of modeling style rules that you should apply when modeling different UML diagrams.*

In Closing

We have discussed five best practices for applying UML in software modeling. That should be enough information to digest for this time. But wait for the upcoming whitepapers – we are going to present many more best practices!

About the Author

Darius Šilingas, Ph.D.
Principal Trainer of MagicDraw
darius.silingas@nomagic.com

Darius has been working as professional software developer starting from 1997. He took Java developer's role in multiple international projects. Currently, Darius works as an analyst and principal trainer in MagicDraw UML modeling tool development team. Darius has developed multiple modeling case studies and run MagicDraw & UML training courses worldwide. He also taught commercial training courses on Requirements Management with UML, Software Design with UML, and Java/J2EE Programming. Darius received PhD from Vytautas Magnus University, where he teaches courses Software Development Methodology and Software Engineering. Additionally, he holds several professional certificates from Microsoft, Sun, and OMG. Darius participates actively in various events and presents talks on UML and other software development topics. He holds the following professional certificates:

- Microsoft Certified Professional
- Sun Certified Programmer for the Java 2 Platform 1.4
- OMG-Certified UML Professional Fundamental
- OMG-Certified UML Professional Intermediate

Services Contacts

Please contact us at services@nomagic.com for getting more detailed information about No Magic services, training courses, scheduling possibilities, or if you have any specific needs for your company.

No Magic, Inc.
Texas
7304 Alma Drive, Suite 600
Plano, TX 75025
Phone: +1 214 291 9100
Fax: +1 214 291 9099
www.nomagic.com
www.magicdraw.com

No Magic European Sales Office
UAB "Baltijos programine iranga"
Savanoriu pr. 363
LT-49425, Kaunas, Lithuania
Phone: +370 37 324032
Fax: +370 37 320670
http://www.bpi.lt/en

No Magic Asian Development Center
No Magic Asia Ltd.
Rama IX Road, Bangkapi, Huaykwang
Bangkok 10310, Thailand
Phone: +66-2-7170250
Fax: +66-2-7170251
http://www.nomagicasia.com