



Sirius

Eclipse Sirius

BEST PRACTICES

Summary

Project Organization

3

Development Methodology

5

Organizing the Viewpoint Specification Model(s)

6

Specifying Representations

7

Performance and Scalability Considerations

10

Specifying Tools

12

Properties View

15

Creating Intuitive Editor

18

Industrializing the Designer

22

These Best Practices mainly relate to Sirius usage. They are up-to-date regarding Sirius 5.0.

Project Organization (1/2)

Start the name of your project with the name of the domain model

If your domain model (metamodel) is named **com.company.domain**, the convention is to name the plug-in project containing your designer as **com.company.domain.design** if you have a single one, and **com.company.domain.viewpoint1.design**, **com.company.domain.viewpoint2.design**, etc. if you have many Viewpoints and want to organize them further.

Keep your Viewpoint Specification independant from the UI

For advanced designers, you may need customizations going beyond what is possible to do with a VSM only, like new wizards, Eclipse views, or other extensions. It is recommended in that case to put these extensions in a different project, and to name it **com.company.domain.design.ui**.

Explicitly state the dependencies in your MANIFEST.MF: for domain model and interpreter in particular

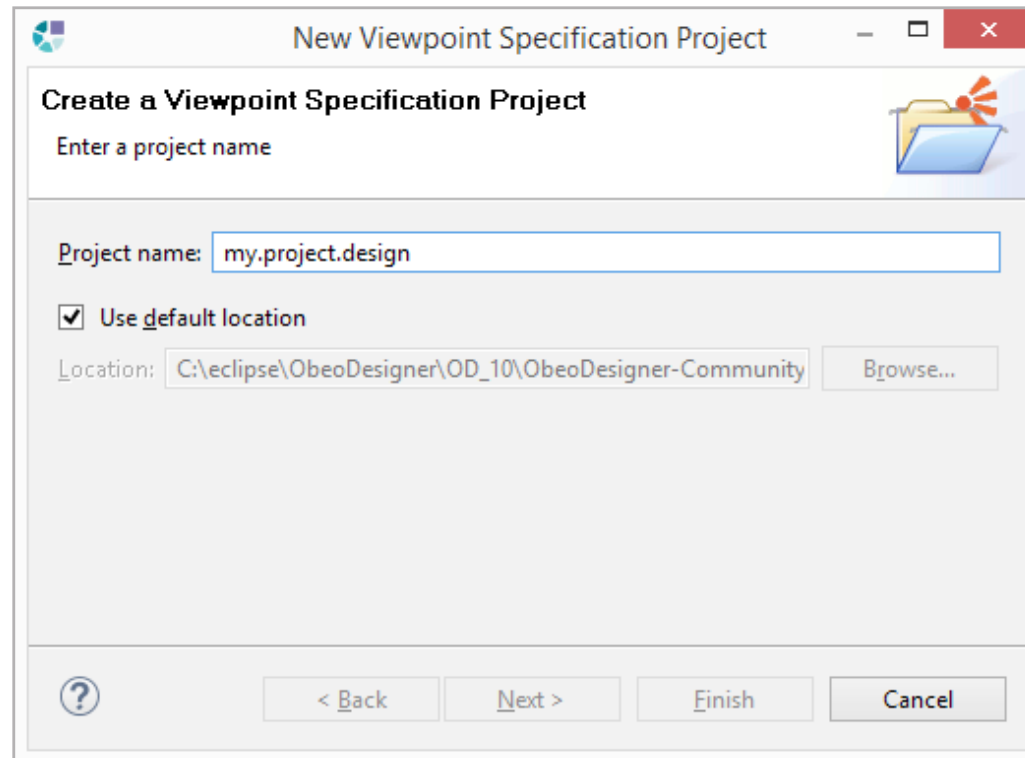
Sirius supports several query languages (var, feature, service, AQL, etc.), but all these are brought by optional dependencies. To avoid potential problems when deploying your designers, add an explicit dependency from your Viewpoint Specification Project to the plugins implementing the languages your VSM depends.

Also explicitly state for the same reason in your MANIFEST.MF any dependency to meta-models used by your designer (e.g. UML, BPMN, ...).

Project Organization (2/2)

Use the Viewpoint Specification Projects wizard

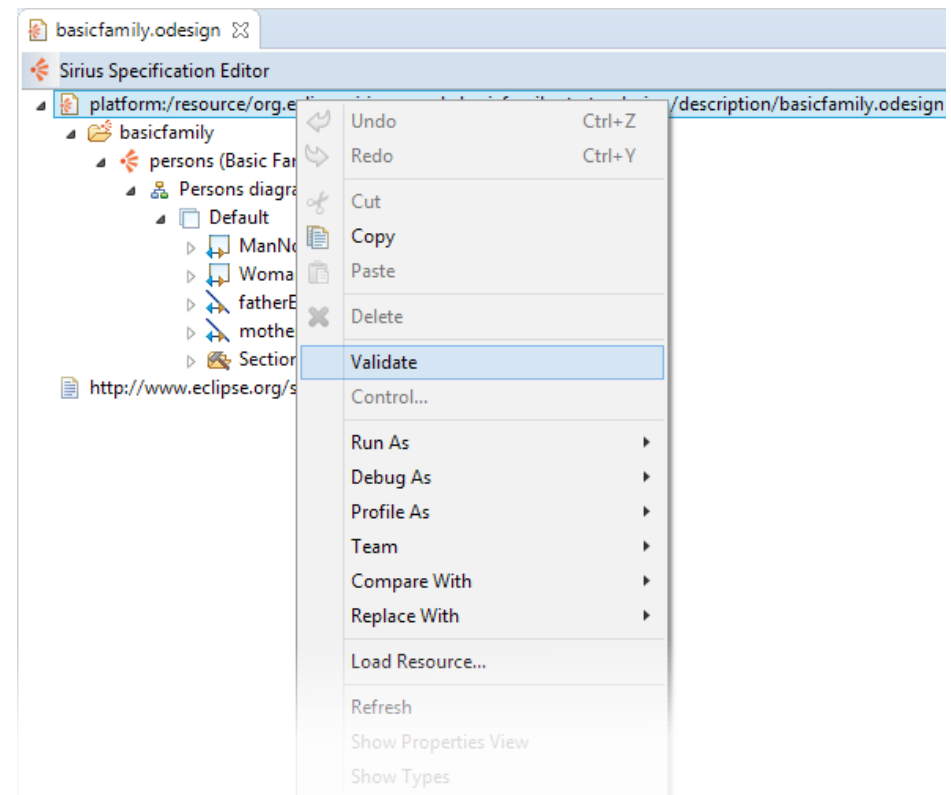
Although this is not strictly required, you are encouraged to use the « **New > Viewpoint Specification Project** » wizard to initialize your specification project: it sets up everything needed to have a VSM ready to deploy to end-users.



Development Methodology

Develop your VSMs dynamically and incrementally using an example model. Sirius supports dynamic changes on VSMs even while the designer is in use: take advantage of this feature to develop your designers much faster. Start with a sample semantic model and a minimal VSM (for example showing an empty diagram on some kind of semantic element). Create an initial representation (diagram, table, or tree), which may be empty, and open the odesign file alongside the representation editor. Whenever a modification made on the odesign file is saved, it is immediately reflected on the representation.

Sirius tries its best to handle at runtime most kinds of changes in the odesign, though a few cases are not handled. In these rare cases, if it seems like the update was not handled correctly, try to close and re-open the diagram. If it still doesn't work as expected, delete the aird file, which stores representation data, and create a new one.



Use the validate button

Validating a viewpoint specification model before testing can avoid unnecessary headaches. The absence of errors and warnings during validation can not guarantee the absence of bugs in your designer, but if there are unfixed errors/warnings, they will most probably cause bad behaviors.

Organizing the Viewpoint Specification Model(s)

Naming conventions

Give an explicit identifier to your mappings to facilitate their reuse. You can provide the label to be presented to users (in the case of tools for example), so do not hesitate to make the identifier technically precise and meaningful for specifiers.

Do not change mapping identifiers once a designer has been deployed to users: it is used by representation instances (e.g. diagrams) making use of this mapping. These identifiers are part of your designer's API and are not visible to the end user. Do not change them once they have been used by clients or that will break their existing representations, forcing them to recreate/regenerate them. The label however, can be changed safely.

Keeping Imported Java Classes number low (Java Extension)

Every class imported in the .odesign will lead to a classpath search. Depending on the extent of your project dependencies this can take time. As such it's generally a good idea to limit the number of imported classes low.

It can get hairy to track what services should be made available for the different viewpoints. A pattern you can use is to import a single class for each Viewpoint (for instance DesignServices or DocumentationServices) and use Java inheritance to match the Viewpoints relationships.

Use naming conventions for mappings and tools

Naming conventions help identifying mapping and tools while designing Viewpoints. Choose a naming convention and stick to it. You could for instance use an acronym for every diagram and prefix the mappings ids with this acronym.

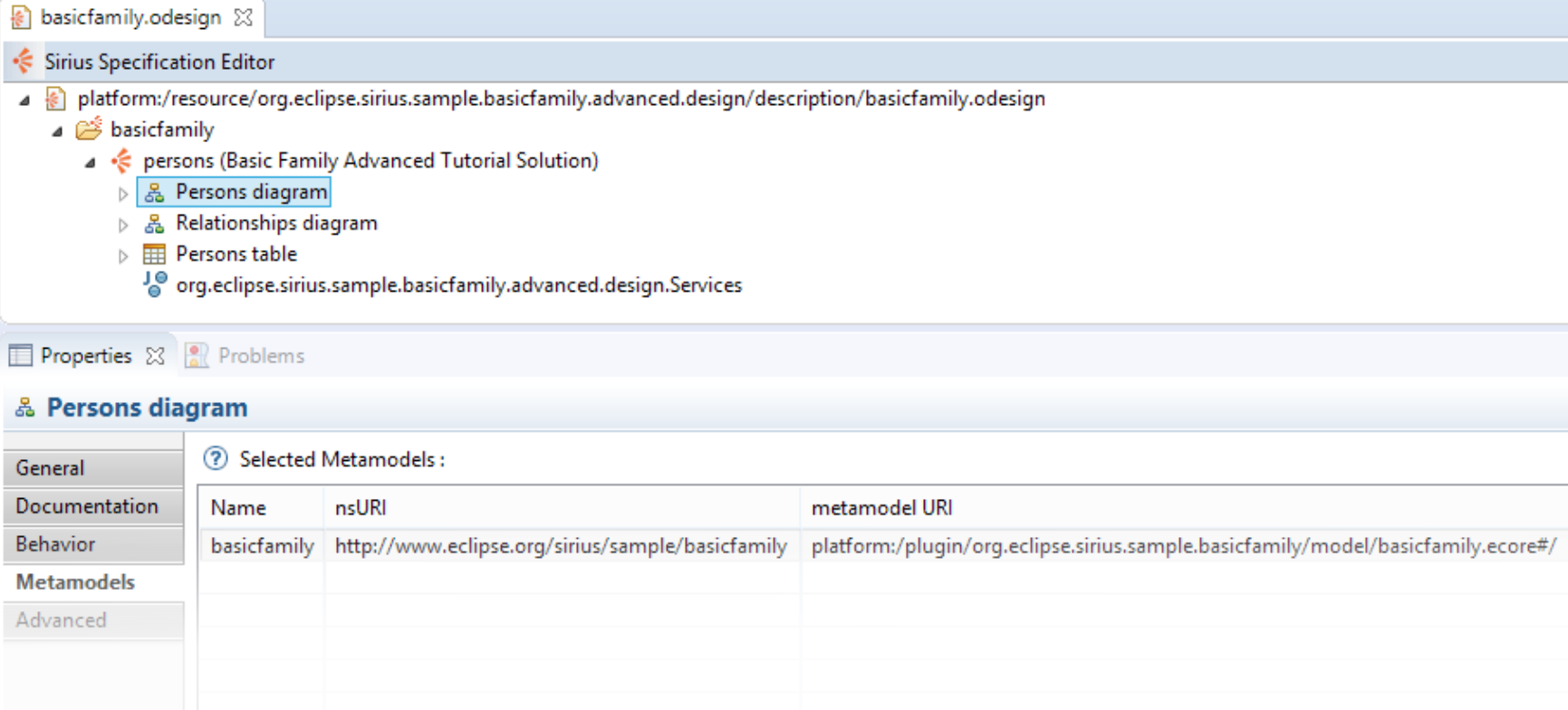
Group your tools into sections

You could for example group all the tools not visible in the palette (like renaming, drag and drop, edge reconnexion, etc.) into a 'usability' section.

Specifying Representations (1/3)

Link Representation descriptions to the Ecore model(s) they operate with

The meta-models a representation is working with have to be explicitly specified. This is done by setting the Metamodels property on your representation descriptions. Doing so will improve the specifier experience by enabling code completion in the various expression fields and VSM validation.



The screenshot shows the Sirius Specification Editor interface. The top bar displays the file name 'basicfamily.odesign'. Below it, the 'Sirius Specification Editor' tab is active. The project tree on the left shows the following structure:

- platform:/resource/org.eclipse.sirius.sample.basicfamily.advanced.design/description/basicfamily.odesign
 - basicfamily
 - persons (Basic Family Advanced Tutorial Solution)
 - Persons diagram (selected)
 - Relationships diagram
 - Persons table
 - org.eclipse.sirius.sample.basicfamily.advanced.design.Services








The 'Persons diagram' is selected, and the Properties panel is open. The 'Selected Metamodels' table is visible, showing the following data:

| Name | nsURI | metamodel URI |
|-------------|--|---|
| basicfamily | http://www.eclipse.org/sirius/sample/basicfamily | platform:/plugin/org.eclipse.sirius.sample.basicfamily/model/basicfamily.ecore# |
| | | |
| | | |
| | | |
| | | |

Specifying Representations (2/3)

Style customizations

Instead of using conditional styles to change the style itself according to a condition, you can use style customizations to change a style properties according to a condition. Unlike conditional styles, style customizations are cumulative. It will therefore be more efficient will be more efficient than conditional styles if using this constructions you can avoid to test all the combinations of predicates.

- ▲  Default
 - ▶  Section Tools
 - ▲  **Style Customizations**
 - ▲  Style Customization `aql:self.oclAsType(basicfamily::Person).children->size()=0`
 -  Property Customization (by selection) `labelColor`
 - ▶  Style Customization `aql:self.oclAsType(basicfamily::Person).children->size()>1`
 - ▶  Style Customization `aql:self.oclAsType(basicfamily::Person).children.children->size()>0`

Specifying Representations (3/3)

Order conditional styles

A mapping can have multiple conditional styles. By default, these styles are simply ordered by their creation sequence.

Being aware that checking a style's condition can be costly (depending on what the expression does), it generally is a good idea to order your conditional styles starting with the ones you expect will be used more often in order to minimize the number of expressions to evaluate.

However since style customization has been introduced, it is now recommended to use it rather than conditional styles.

Create a representation description for reuse purpose

Defining different kind of representations can easily lead to duplicate mappings or tools among them. As duplicating is usually not a good idea regarding maintenance and scalability, it is a good practice to define an abstract representation for the purpose of defining generic mappings and tools to be reused in concrete representations.

This is done by setting the pre condition of the representation to false, thus not allowing the user to see it.

Style customizations

Instead of using conditional styles to change the style itself according to a condition, you can use style customizations to change a style properties according to a condition. Unlike conditional styles, style customizations are cumulative. It will therefore be more efficient will be more efficient than conditional styles if using this constructions you can avoid to test all the combinations of predicates.

Performance and Scalability Considerations (1/2)

Measure

Take a few minutes to test your designer with a big model (copy/pasting an actual model works great for this). Activating the Sirius Profiler and showing its dedicated view will let you see where time is spent, allowing you to focus on critical matters.

Use the most appropriate query language

Use AQL. AQL is faster than Acceleo or OCL. If you are concerned with high performance, the « **feature:** », « **var:** » and « **service:** » languages are even faster, presenting no overhead. These languages are built in Sirius and doesn't introduce any extra dependency.

On the other hand, using different query languages makes the Viewpoint Specification less homogenous, not helping maintenance.

Avoid eAllContents()

The eAllContents() operation is very attractive when specifying a Viewpoint. However, keep in mind it implies a full traversal of the model contents! It is anodyne when working on small models but can drastically impact performance on big models. Prefer using more specific queries traversing smaller parts of the model.

Avoid empty semantic candidate expression

A semantic candidate expression left empty evaluates as 'all models elements conforming to the mapping's domain class', thus equivalent to a eAllContents() call. This is -a convenient behavior easing the start of a new representation description but we encourage not to leave it empty.

We encourage to enter an expression (or a Java service) leveraging the current diagram content (using the **\$diagram** variable) to find what elements are already displayed and browse the model from there (using the cross referencer when needed) to retrieve the sub mapping candidates. The **following commit in UML Designer shows an example of how it can be done.**

Don't use EcoreUtil.delete(), never

EcoreUtil.delete() will browse the whole resource set to remove potential dangling references. This gets slower when your model grows.

Starting with Sirius 1.0 the runtime itself cares for the dangling references **EcoreUtil.remove()** should be enough. If not, make sure you uses the session Semantic Cross Referencer if you need to traverse inverse references.

Performance and Scalability Considerations (2/2)

Use `eInverse()`

Relationships between elements in EMF are by default uni-directional (A points towards B through reference r). At some point, you will most probably need to traverse relationships at reverse, needing to know what elements points to some other, or what elements point to some other through some specific relationship.

A possibility would be to iterate on all elements in the models, testing for each if it points to the concerned element. That's obviously inefficient and should be avoided.

Sirius can help in that matter as it keeps track of that information. Use the `eInverse()` service in your expressions to access that information at no cost!

EMF Switch global behavior

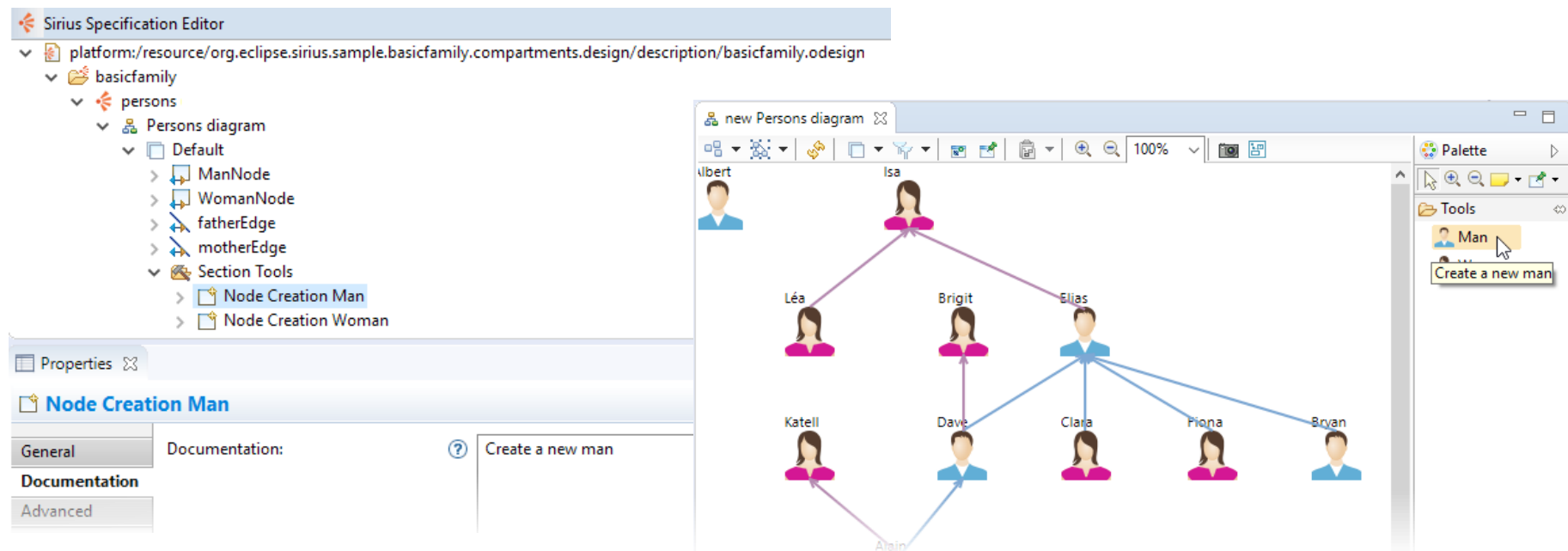
EMF generates a switch for your metamodels, that can be used to dispatch some process depending on the type. This is very useful for complex metamodels comprising many inheritance relationships.

For example, it is useful to describe in the odesign the call of the same `displayLabel()` service for each mapping. Then, this service use the emf switch to dispatch the label edition for each metaclass. Thus, you can enjoy the java inheritance to capitalize label edition and the java compilation to manage metamodel evolutions.

Specifying Tools (1/3)

Fill in the « Documentation » properties

Specify the documentation of the tools visible in the palette: Sirius will provide a tooltip to the user containing this documentation. It will facilitate the adoption of your modeling tool. In general, you should also document the other elements of your VSM in order to help the maintenance of your solution.



Specifying Tools (2/3)

Always start with a Change Context

Some of the operations available for tool specifications only allow for a single sub-operation. This is often the case for the « root » operation of the tool itself (Begin). To be able to add several operations, you can use the « Change Context » operation (aka « Go To ») with a no-op target expression like **var:self**. Inside it you can then add multiple operations which will be executed in sequence. You can think of it as the equivalent of a **block { ... }** in a programming language.

- Section Tools
 - Node Creation Man
 - Node Creation Variable container
 - Container View Variable containerView
 - Begin
 - Change Context var:container
 - Create Instance basicfamily::Man
 - Edge Creation Father
 - Source Edge Creation Variable source
 - Target Edge Creation Variable target
 - Source Edge View Creation Variable sourceView
 - Target Edge View Creation Variable targetView
 - Begin
 - Change Context var:source
 - (x)= Set father
 - Reconnect Edge reconnectFather
 - Source Edge Creation Variable source
 - Target Edge Creation Variable target
 - Source Edge View Creation Variable sourceView
 - Target Edge View Creation Variable targetView
 - Element Select Variable element
 - Begin
 - Change Context var:element
 - (x)= Set father

Specifying Tools (3/3)

Keep the usage of imperative language simple

VSMs should only specify graphical and behavioral features related to the GUI. The imperative language that the Viewpoint specification provides is not aimed to implement complex logic. Business complexity should instead be externalized out of the VSM. Java services are to be used to program this logic, the viewpoint imperative language only calling these.

When implementing some logic in the VSM, ask yourself if it could be placed in the metamodel itself (using **@generated NOT** annotations) or maybe mutualized in some services class. Applying this pattern will help not duplicating the logic implementations among your designers and other EMF based tools.

Use sub-variables

You can define sub-variables computed from one of the pre existing tool variables. When you notice you are copy pasting many times the same expression, it probably means you should use a sub-variable.

Use two reconnect edge tools instead of a single one

The reconnexion tool definition allows the handling of both ends reconnexion. However, specifying such a tool can be touchy and error-prone. We recommend not using this option unless you have good reasons. Most of the case, it is easier to separate the source and target reconnexion in two separate tools, thus also facilitating maintenance.

Always use the NS (namespace) prefix

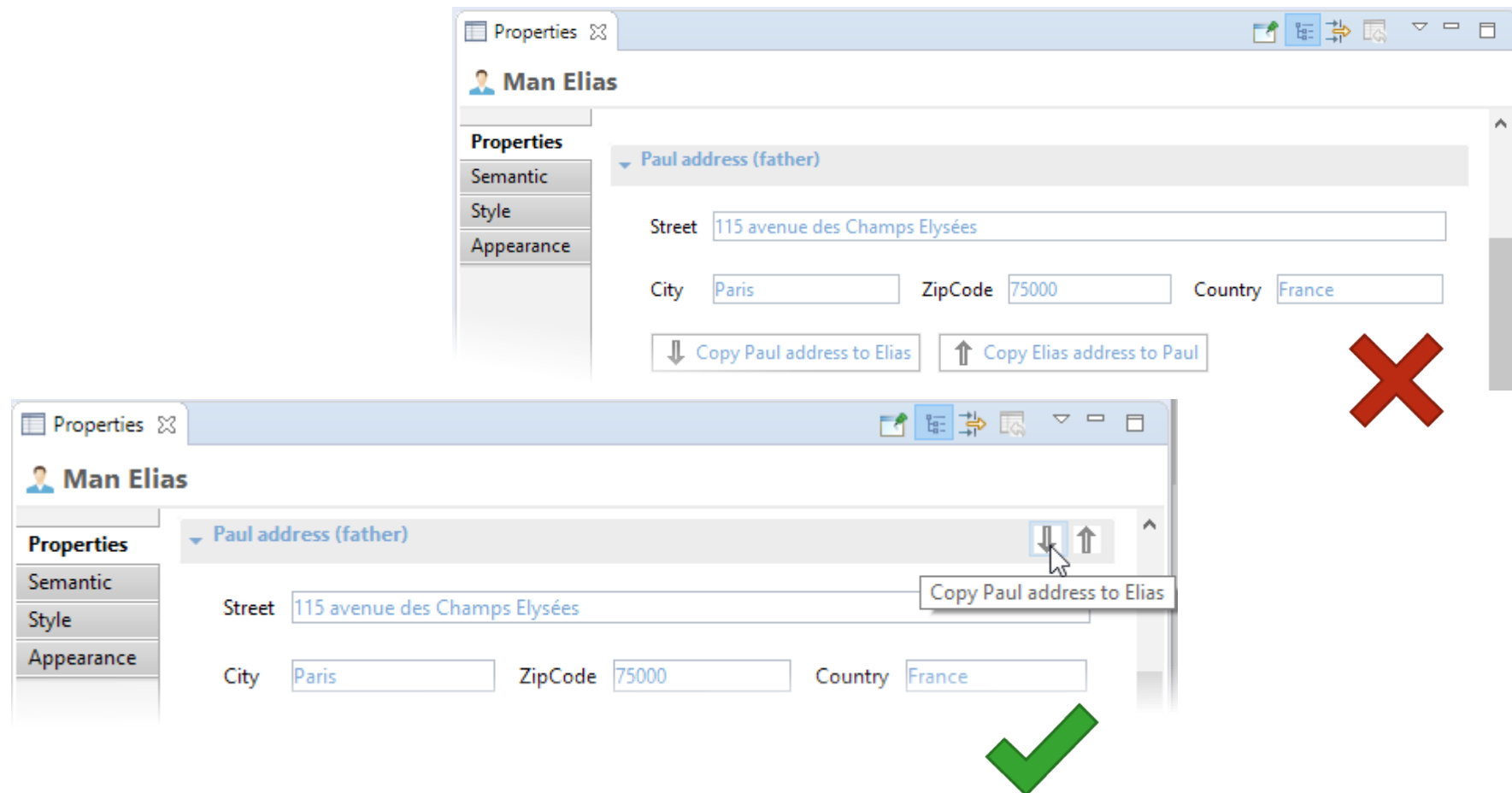
When specifying a Domain class (in a green field), always prefix its name with the metamodel NS prefix (for example **uml.Attribute** instead of just **Attribute**). This will prevent eventual conflicts with another metamodel that could define a class of the same name.

Domain Class*:  **basicfamily::Family**

Properties View (1/3)

Use toolbar actions to simplify your user interface

Instead of multiplying buttons in the groups of your property views, you can create toolbar actions on pages and groups with meaningful icons to offer the same feature in a less cluttered user interface.



Properties View (2/3)

Use the menu to create a new widget from your domain class

When you need to create a new widget manually, don't start with a blank widget if you want to link it with a property from your meta-model. If the domain class of the Group is properly configured and if your meta-model has been registered in the Properties View element, Sirius can create widgets preconfigured to edit the properties of your domain class with the menu « **New Widget From Domain Class** ».



Properties View (3/3)

Lower the amount of duplicates in your odesign with inheritance and composition

In the case you want to add the ability to open a dialog showing the properties of an object when the user will double click on it in a diagram, while showing those same properties in the properties view, do not duplicate all the elements used to display your properties. Instead, create some common pages or groups in a category and extends or reuse them in your properties view, dialogs and wizards. This way, you can have the same user interface in a dialog used by a creation tool, a double click and even your property view easily.

Use indented page to organize the complexity of the information

With indented pages, you can create sub-tabs in your properties view where you can let the user edit advanced properties, for example.

Use categories to organize your properties in the odesign

We have introduced a concept of category to let you organize your own pages and groups in your properties view. Those categories are not used by the runtime so you can use them as you want without any impact.

Leverage the `EditingDomainServices` class to simplify your expressions

Note that you can add the class **`org.eclipse.sirius.ext.emf.edit.EditingDomainServices`** in your odesign and its bundle **`org.eclipse.sirius.ext.emf.edit`** to your **`MANIFEST.MF`** to have access to a collection of useful services for all your expressions. With this class, you will be able to leverage the power of EMF Edit with services for the label provider, property descriptors, command framework and much more.

Use dynamic mappings to handle most of your meta-model

With dynamic mappings, you can easily define rules specifying when a widget will be used. This way, you can define that a text widget should be used for all the attributes with the type string and a checkbox for all the booleans.

Creating Intuitive Editors (1/4)

There is no such thing as a globally intuitive editor. An editor shall provide a good user experience for a specific user group. Your first task should then be to answer the question what are my target users?

A simple and quick methodology can be applied:

- Starts by defining 2 or 3 **Personas**, name them, and write down their background.
- Write down a situation where the personas are using your tool. List the tasks the personas are supposed to achieve, list their expectations regarding the tool during this situation.
- Use the persona's background to justify the choices of what should be emphasized in the diagram at this point and what colors or shapes should be used.

Critical questions you need to ask, get answer and motivations:

- What is the diagram synchronization strategy?
- How is the tool giving feedback to the persona? When?
- How critical aspects are emphasized? (use of boldness, colors)
- What are the Viewpoints, Layers, how do they match the persona activities?

Have a look on the **corresponding work that have been done for EcoreTools 2.0**.

Here is also a blog post describing features you might want to consider in your tooling.

Unless you have some specific requirements going against it, some general guidelines should be applied:

Provide basic tools

Always define direct edit, reconnect, drag and drop and navigation tools. Users are expecting this basic behavior.

Provide startup representations for the main objects of your model

A model is most commonly composed of a few main objects held by a root object. A startup representation showing up automatically (with « Initialization » and « Show On Startup » properties set to true) and presenting these main objects is a good starting point for the user to explore or modify the model.

Another frequent pattern is to use a Dashboard representation giving the advantage of supporting a methodology in addition of being a starting point

Creating Intuitive Editors (2/4)

Use color as a support for the semantic and as a way to give feedback

Different kinds of objects should be easily distinguished.

Use colors consistently

Use as much as possible the same colors and forms for the same objects visible on different representations. As the border color of a shape, select the darker version of the shape color: for example if the shape color is green, use dark_green as border color. In a gradient, use white as one of the two colors.

Clean your SVG files

For a SVG image on a container node: add **viewBox = « 0 0 width height »** (replace width and height) and **preserveAspectRatio = « none »** in **<svg>** attributes.

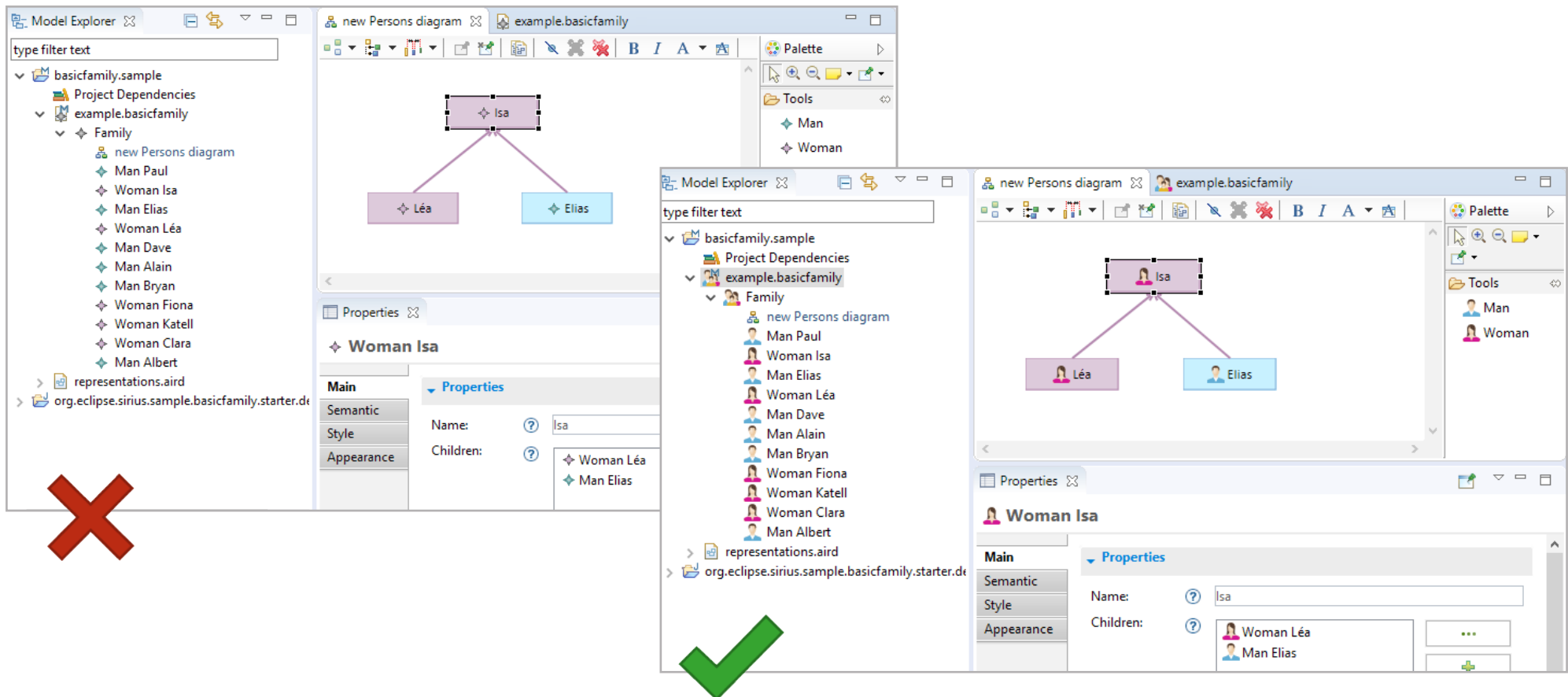
Avoid prefixing creation tools with « Create »

In most cases the tools allow to create new objects.

Creating Intuitive Editors (3/4)

Add icons for each metaclass

Nothing is worse for a graphical designer than to keep the default auto-generated **.edit** icons. Use specific icons (**take a special attention to the the license though, and make sure to reference the origin source of those icons**), at least in the .edit project this icons are used by Sirius in every corner of the UI.



Creating Intuitive Editors (4/4)

Add dialogs to your creation tools if you need user input

When several properties (in addition to the name) need to be initialized by the user at the creation of a new element, provide a dialog during the execution of a creation tool to ask the user for some input

The dialog box is titled "New" and contains the instruction "Please complete the properties of this new person". It is divided into two main sections: "Main" and "Address".

Main

Name

Address

Street

City ZipCode Country

At the bottom of the dialog, there is a help icon (a question mark in a circle) on the left, and two buttons, "Finish" and "Cancel", on the right.

Industrializing the Designer

- Provide initialization wizards
- Define tests
- Test the deployed version of the designer
- Bundle examples with the designer
- Check branding, licenses headers and copyrights
- Setup an automatic build environment
- Write a Getting Started guide

Rely on the best Sirius professionals!



Training



Consulting & Coaching



Custom Development



Support & Maintenance

Discover Our Services