



Service Robotics Ulm
autonomous mobile service robots

Hochschule Ulm



University of
Applied Sciences

SmartMDSD Toolchain User Manual

Servicerobotik Ulm
University of Applied Sciences Ulm
Prittwitzstr. 10
D-89075 Ulm
Germany
<http://www.servicerobotik-ulm.de>

SmartMDSD Toolchain User Manual:

Copyright © 2016 Sandra Frank, Dennis Stampfer, Christian Schlegel

Servicerobotik Ulm
University of Applied Sciences Ulm
Prittwitzstr. 10
D-89075 Ulm
Germany

The CSS Stylesheet used for HTML output is based on the Debian Reference Manual CSS.

Continuous updates of this document are available through <http://www.servicerobotik-ulm.de>:

- *PDF* [<http://www.servicerobotik-ulm.de/toolchain-manual/manual.pdf>]
- *Browsable HTML* [<http://www.servicerobotik-ulm.de/toolchain-manual/html>]

Document History

For use with SmartMDSD Toolchain v2.10

March 31th 2016

- First version.
-

Table of Contents

1. Introduction	1
1.1. Fundamentals	1
1.1.1. The SmartSoft World	1
1.1.2. Components	2
1.1.3. Services	2
1.1.3.1. Communication Objects	2
1.1.3.2. Communication Patterns	2
1.2. Resources	3
1.2.1. Online Resources	3
1.2.2. Further Reading	3
2. Using the SmartMDSD Toolchain	4
2.1. Introduction	4
2.1.1. Installation and Requirements	4
2.1.2. Development Workflow	4
2.1.3. Working with the SmartMDSD Toolchain and SmartSoft Framework	4
2.1.3.1. SmartMDSD and SmartSoft on the filesystem	4
2.1.3.2. Starting the SmartMDSD Toolchain for the First Time	5
2.1.3.3. SmartMDSD Eclipse Projects	8
2.1.3.4. Version Control Considerations	9
2.2. System Design View	9
2.2.1. Project Structure	9
2.2.2. Communication Objects	10
2.2.2.1. Modeling Communication Objects	10
2.2.2.2. Generated files	13
2.2.3. Parameter Sets	14
2.2.4. Compile SmartSoft Communication/Coordination Repositories	16
2.2.5. Documentation	17
2.3. Component Development View	17
2.3.1. Component Projects	18
2.3.2. Component Modeling	19
2.3.2.1. Component Hull	19
2.3.2.1.1. Communication Patterns	20
2.3.2.1.2. SmartTask	23
2.3.2.1.3. SmartComponentMetadata	24
2.3.2.2. Component Parameters	24
2.3.2.2.1. SmartParameterMaster	24
2.3.2.2.2. SmartParameterSlave	24
2.3.2.2.3. SmartComponentParameter	24
2.3.2.2.4. Parameter Documentation	27
2.3.3. Component Implementation	27
2.3.3.1. Generated Files	28
2.3.3.2. Start Services and Tasks	30
2.3.3.3. Using Communication Objects	30
2.3.3.4. Using Services	31
2.3.3.5. Status Codes	33
2.3.3.6. Component Wide Variables	33
2.3.3.7. Using Parameters Within the Component	34
2.3.4. Compile SmartSoft Component Projects	35
2.3.4.1. Add Additional Libraries	35
2.3.4.2. Add Compiler Flags	36
2.3.4.3. Add Your Own Source Files	36
2.3.5. Component Documentation	37
2.4. System Composition View	38
2.4.1. System Composition Project	39
2.4.2. System Configuration	39

2.4.2.1. System Configuration Model	39
2.4.2.1.1. Change Connections	40
2.4.2.1.2. Delete Components From the Model	40
2.4.2.2. Component Instance Configuration	41
2.4.3. System Deployment	42
2.4.3.1. Deployment model	42
2.4.3.2. Code Generation	43
2.4.3.3. Target Considerations	44
2.4.3.4. Deploying Additional Files	44
2.4.3.5. Start-Stop-Hooks	44
2.4.3.6. Predeploy Infrastructure	44
2.4.3.7. Deploying the Application	44
2.4.4. Running the Application	45
2.4.4.1. Running the Application from Toolchain	45
2.4.4.2. Running the Application without the Toolchain	46
2.4.4.3. Component Output and Log Files	46
2.5. Tips and Tricks	46
2.5.1. SmartSoft Full Build of Source Tree	46
2.5.2. SmartSoft and the RaspberryPi	47
2.5.2.1. Preconditions	47
2.5.2.2. Step by Step Instructions	47
2.5.3. Delete Model Elements	48
2.5.4. Common Error Messages	48
2.5.4.1. Component Development View	48
2.5.4.2. System Composition View	49
3. Tutorials	51
3.1. Video Tutorials	51
3.1.1. Tutorial 1: Modeling of Communication Objects	51
3.1.2. Tutorial 2: Definition of a Parameter Set	51
3.1.3. Tutorial 3: Component Development	51
3.1.4. Tutorial 4: System Configuration and Deployment Model	52
3.1.5. Tutorial 5: Deploying and Running an Application	52
3.1.6. Tutorial 6: Deployment of components along with additional files	52
3.2. Step by Step: Robot navigation	53
3.2.1. Introduction	53
3.2.2. Component Development (SmartKeyboardNavigation)	53
3.2.3. System Composition	58
3.2.3.1. System Composition 1: Simple Scenario	58
3.2.3.2. System Composition 2: Adding obstacle avoidance	60
Bibliography	63

List of Figures

1.1. SmartSoft Component, Communication Objects and Services	1
1.2. Available communication patterns.	2
2.1. Start toolchain and select new workspace	5
2.2. Start toolchain and select new workspace	6
2.3. Select the SmartSoft perspective to set-up up eclipse for efficient SmartSoft modelling use.	6
2.4. The started SmartMDSD toolchain.	7
2.5. Import projects into the workspace, in this case communication/coordination repository projects projects.	8
2.6. Toolchain with imported communication objects repository projects.	8
2.7. Create a new Communication/Coordination Project	9
2.8. Structure of communication/coordination projects	10
2.9. Model Communication Objects	11
2.10. Add custom functions	14
2.11. Model Parameter Sets	15
2.12. Compile a communication/coordination project	17
2.13. Documentation of communication objects	17
2.14. Create a new component project	18
2.15. Structure of Component Projects	19
2.16. Modeling Component Hull	19
2.17. Select Communication object	23
2.18. Select handler	23
2.19. Model Component parameter	25
2.20. Code generation	28
2.21. Start services	30
2.22. Example implementation of a task using a service. [4]	31
2.23. Add Component wide variables	34
2.24. Compile SmartSoft Component Project	35
2.25. Information from the documentation and component model is transformed to a complete documentation (right) for later system integration which assists the system integrator during composition. [4]	38
2.26. Create a new deployment project	38
2.27. Structure of System Composition Projects	39
2.28. Modeling System Configuration	40
2.29. Delete Component from Model Explorer	40
2.30. Delete Project from Java Build Path	41
2.31. Configure component instance	42
2.32. Selecting a utilized component instance.	43
2.33. Modeling System Deployment	43
2.34. SmartSoft Deployment Code Generator	44
2.35. Running the application from the toolchain	45
2.36. Global Scenario Control	46
2.37. Console tab of the SmartMDSD Toolchain	48
3.1. SmartKeyboardNavigation	54
3.2. KeyboardInputTask task with its timing parameters.	54
3.3. Adding parameters	57
3.4. System Configuration	59
3.5. Deployment	60
3.6. System Configuration	61
3.7. Deployment	62

Chapter 1. Introduction

This is the SmartMDS Toolchain User Manual. It contains information how to use the SmartMDS Toolchain for modeling, implementing and integrating SmartSoft software components. The SmartMDS Toolchain guides through the development workflow while applying the methods and principles from the SmartSoft World as they are for example used within the FIONA Project.

The initial version of this document was published on March 30th 2016 as deliverable of the ITEA2 "FIONA" Project (Framework for Indoor and Outdoor Navigation Assistance, <http://www.fiona-project.eu>): Deliverable D2.4.1 "Handbook for Integrating Basic Services and Interface Components". The presented document is the continuously updated version.

The manual is intended for users of the SmartMDS Toolchain. For scientific contributions, please refer to publications of SmartSoft and the SmartMDS Toolchain. Some relevant publications are also highlighted within the section 1.2.2.

The Introduction first sketches some very brief basics of the SmartSoft World. In chapter two, it describes the SmartMDS Toolchain from its three main development perspectives: system design, component development and integration by composition. The third part describes tutorials (in written form and referenced video screencasts) to guide the reader through all these steps using a practical real-world example. All parts of the handbook can be read in sequential order.

1.1. Fundamentals

SmartSoft nowadays is an umbrella term for abstract concepts (such as a systematic development methodology, best practices and implementations (reference implementations, a set of reusable components) to build robotics systems. The SmartMDS Toolchain is an Integrated Development Environment (IDE) for robotics software development. It seamlessly integrates into the world of SmartSoft by realizing the concepts of SmartSoft, thereby making them accessible to its users. [4]

1.1.1. The SmartSoft World

The SmartSoft World consists of several elements:

- *SmartMARS MetaModel*. It defines the structure for communication objects, a set of communication patterns, the structure of services and components, the structure for system configuration and deployment.
- *SmartSoft Framework and implementation*. In its current state, two exchangeable reference implementations (ACE and CORBA middlewares). Execution containers for several platforms and operating systems. The SmartMDS Toolchain currently supports SmartSoft/Ace.
- *SmartSoft MDS Toolchain*. An Integrated Development Environment (IDE) for robotics software development that supports the separation of roles. The toolchain covers the development process of modeling communication objects, components and systems.

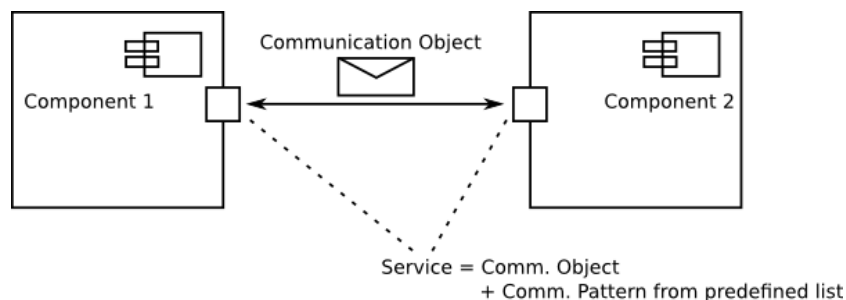


Figure 1.1. SmartSoft Component, Communication Objects and Services

1.1.2. Components

Components are technically implemented as processes. A component can contain several threads and interacts with other components via predefined communication patterns and communication objects. The component hull provides a stable interface between the internal structure of a component (inner view of a component) and its outside view (services, for the system integrator). Within components, component developers find a structure to implement algorithms, reuse libraries and communicate with other parts of the system through services. The SmartMDSD Toolchain assists in modeling and implementing components. A set of existing components (SmartSoft SVN repository) is available for immediate reuse and composition to applications from within the SmartMDSD Toolchain.

1.1.3. Services

A service is a combination of communication object(s) and communication pattern as defined by SmartSoft. A communication pattern connects the externally visible service (the stable outer view) with the internally visible set of access methods (the stable inner view) for this service. Technically, generic predefined communication patterns become services by binding one given communication pattern with communication object(s).

1.1.3.1. Communication Objects

Communication objects define the data structure (content) to be transmitted via a communication pattern between components. Communication objects are ordinary C++-like objects decorated with additional member functions for data access and internal use by the framework.

Communication objects are always transmitted by value to avoid fine grained intercomponent communication each time an attribute is accessed. Furthermore, object responsibilities are much simpler with locally maintained objects than with remote objects.

1.1.3.2. Communication Patterns

Communication patterns provide the only link of a component to its external world. They define the semantics and policy of communication. By using a fixed set of communication patterns, the semantics of the services of a component is predefined, irrespective of where the communication patterns are applied. By knowing the communication pattern, the semantics and policy of this particular service of the component is known. This supports and enables separation of roles (system integrator can rely on the known pattern) and system composition (services become exchangeable) where new applications can be composed by reusing already existing software building blocks.

Patterns for communication		
Send	Client/server	One-way communication
Query	Client/server	Two-way request/response
Push newest	Publisher/subscriber	1-to-n distribution
Push timed	Publisher/subscriber	1-to-n distribution
Patterns for component coordination and configuration		
Event	Client/server	Asynchronous conditioned notification
Parameter	Master/slave	Run-time configuration
State	Master/slave	Lifecycle management and activation
Dynamic wiring	Master/slave	Dynamic connection wiring
Monitoring	Master/slave	Run-time monitoring of components

Figure 1.2. Available communication patterns.

1.2. Resources

1.2.1. Online Resources

- *Website of Service Robotics Ulm* [<http://www.servicerobotik-ulm.de>]
- *SmartSoft/ACE documentation* [<http://www.servicerobotik-ulm.de/drupal/doxygen/aceSmartSoft/>]
- *SmartSoft and SmartMDSD Toolchain download* [<http://www.servicerobotik-ulm.de/drupal/?q=node/63>]
- *Documentation of available ready-to-use SmartSoft components and communication objects* [http://www.servicerobotik-ulm.de/drupal/doxygen/components_commrep/]
- *Youtube channel: demos and tutorials* [<https://www.youtube.com/user/RoboticsAtHsUlm>]

1.2.2. Further Reading

The following reading resources are recommended in order to understand SmartSoft and its service-oriented component way of thinking as well as the SmartMDSD Toolchain.

- "Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model" [1]
- "Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot" [2]
- "Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns" [3]
- "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software" [4].

This is the most complete publication about the SmartMDSD Toolchain. It describes all links and steps within the SmartMDSD Toolchain.

- A complete list of publications is available at <http://www.servicerobotik-ulm.de/drupal/?q=node/15>.

Chapter 2. Using the SmartMDSD Toolchain

2.1. Introduction

The SmartMDSD Toolchain is based on Eclipse Modeling Tools and uses graphical and textual Domain Specific Languages (DSLs). It uses UML-profiles to implement the SmartSoft robotics meta-model SmartMARS and uses PapyrusUML for graphical modeling. Xtext is used for textual modeling. Additional assistants, validators, checks and glue-logic create the necessary bridges between the DSLs to achieve the overall workflow. Using other Eclipse plugins, e.g. CDT, the component developer can add business logic to the component and implement, for example, algorithms, glue code or reuse libraries. [4]

2.1.1. Installation and Requirements

For running SmartSoft, you need the SmartSoft kernel and ACE. We recommend the script-based installation as described on the *website* [<http://servicerobotik-ulm.de/drupal/?q=node/34>].

To run the SmartMDSD Toolchain we recommend Ubuntu 12.04 LTS and Java OpenJDK 1.6.0_36. A minimum of 4GB RAM is required if you only use few components, but we recommend 6-8GB + for large setups.

We recommend to install the latest version of the SmartMDSD Toolchain using the installation script. If you prefer to install it manually, follow these instructions:

```
# wget http://sourceforge.net/projects/smart-robotics/files/latest/download?source=directory
# tar -xzf SmartMDSD-toolchain-X.X.tag.gz
```

You can start the toolchain by double clicking the extracted binary "SmartMDSD-toolchain-X.X/eclipse" (preferred/standard method).

2.1.2. Development Workflow

The development workflow is divided in modeling communication objects, modeling and implementing components and then using these components to integrate them by composition. The rest of this chapter follows this structure. For a complete description of the development workflow, please refer to [4].

2.1.3. Working with the SmartMDSD Toolchain and SmartSoft Framework

2.1.3.1. SmartMDSD and SmartSoft on the filesystem

After the script installation ACE is installed at `$ACE_ROOT` (typically `/opt/ACE_wrappers`) and SmartSoft is installed at `$SMART_ROOT_ACE` (typically `$HOME/SOFTWARE/smartsoft`). Existing SmartMDSD projects can be found at:

- SmartSoft communication/coordination projects: `$SMART_ROOT_ACE/src/interface-Classes/`
- SmartSoft component projects: `$SMART_ROOT_ACE/src/components/`

- SmartSoft deployment projects: `$SMART_ROOT_ACE/src/deployments/`

The most convenient place to keep new SmartSoft communication/coordination projects is in the directories listed above. If you would like to store SmartSoft communication/coordination projects on an other directory add the following line to your `~/.profile` file:

```
export SMART_PACKAGE_PATH=$SMART_PACKAGE_PATH:<EXTERNALDIRECTORY>
```

Thereby, `<EXTERNALDIRECTORY>` has to be replaced with the directory to the SmartSoft communication/coordination project.

For technical reasons at the moment, you cannot have SmartSoft projects in a directory-tree that leads to other SmartSoft projects. In other words, do not keep a SmartSoft project in a direct hierarchy of another project. Example: it is not allowed to have a SmartSoft Component project at `~/SOFTWARE/smartsoft/`, because there are other projects in `~/SOFTWARE/smartsoft/src/components/`. However, you are fine to keep your custom projects in `~/my_projects/` or `~/SOFTWARE/smartsoft/src/my_components/` or even `~/SOFTWARE/smartsoft/src/components/`. For most cases, e.g. if you are using version control, you will not encounter this problem.

2.1.3.2. Starting the SmartMDSD Toolchain for the First Time

This section will give first time toolchain users an manual on how to set-up the toolchain efficiently.

Start the toolchain by double-clicking the eclipse-executable (recommended) or executing `./eclipse` in a terminal within the directory it has been extracted to. The toolchain will need read and write permissions to all imported eclipse projects, therefore we recommend to start the toolchain with the same user that has checked out the SmartSoft repository / installed SmartSoft.

During start-up the toolchain splash will be displayed stating the version of the toolchain. As is normal for eclipse, the user is asked to select a workspace containing projects and settings. How to organize different workspaces is completely up to the user, in case you have no experience with Eclipse we recommend to keep the multiple workspaces concentrated in one directory e.g. `~/WORKSPACES`.

Starting the toolchain for the first time, a new workspace location could for example be: `/home/$USERNAME/WORKSPACES/SmartMDSD_v2.10`. New directories will be created if necessary.

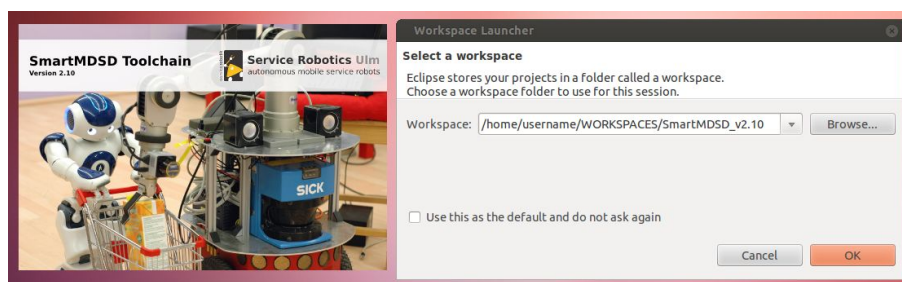


Figure 2.1. Start toolchain and select new workspace

Once the toolchain is completely started and a new empty workspace has been created the "Welcome to the Eclipse Modeling Tools" window will be shown. Closing this inner window will set up the standard (java) eclipse interface.



Figure 2.2. Start toolchain and select new workspace

Eclipse features dedicated window arrangements called perspectives to support different task. The toolchain has its own perspective "SmartSoft". To use this perspective select in menu: Window->Open Perspective->Other... SmartSoft

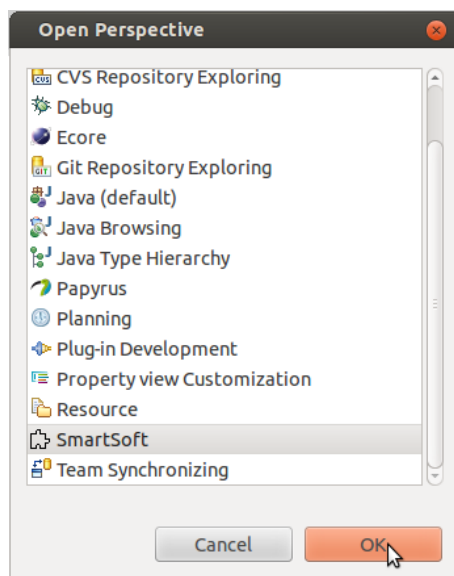


Figure 2.3. Select the SmartSoft perspective to set-up up eclipse for efficient SmartSoft modelling use.

Some toolchain projects make use of automatically triggered code generation, therefore it is necessary that eclipse is configured to build the projects in the workspace automatically. This setting is enabled by default, to ensure this the user can revisit the setting in the menu: Project->Build Automatically

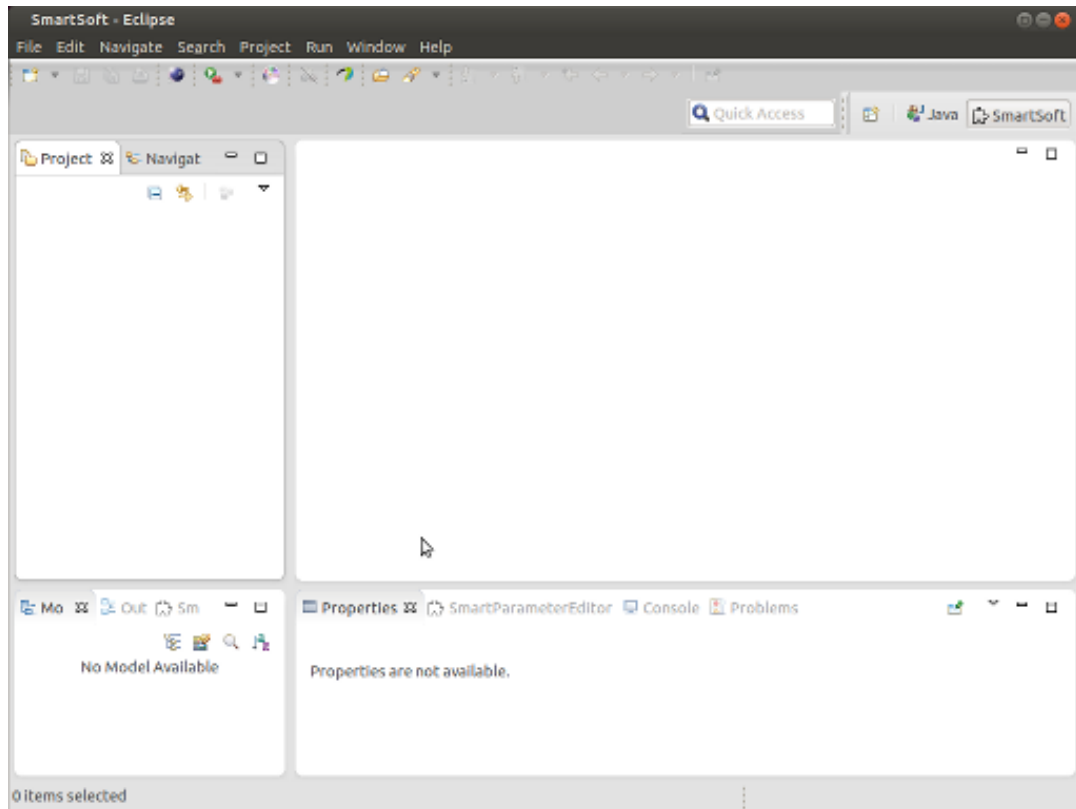


Figure 2.4. The started SmartMDS toolchain.

Given these few steps the toolchain is ready to use. To make use of existing work (components, CommunicationObjects, deployments, etc.) a first step is to import those projects the user wants to reuse, into the workspace.

The different kind of toolchain projects (supporting different roles) depend on each other. Communication objects are wrapped in communication/coordination repository projects, they may depend on other projects of the same type, as communication objects can be nested. Component projects depend on communication/coordination repository projects as they are using communication objects to model the services a component features. Deployment projects depend on component projects as they are used to model a system consisting of components.

Dependent projects, e.g. a communication/coordination repository projects required by a component project, are not imported automatically. Therefore the user has to import all (direct and dependent) used projects.

Since the communication objects are the fundamental building blocks, their projects are typically the first thing to import.

To import projects into the workspace select in menu: File->Import. Within the Import dialog select: General->Existing Projects into Workspace continue with next and select the directory containing the projects to include e.g. /home/username/SOFTWARE/smart-soft/src/interfaceClasses. Eclipse will search recursively for projects from this directory on and will present a list of projects to import. In this example a list of all communication/coordination repository projects that are shipped within the smartsoft repo. Importing all existing communication/coordination repository projects into the workspace is in most cases a reasonable thing to do.

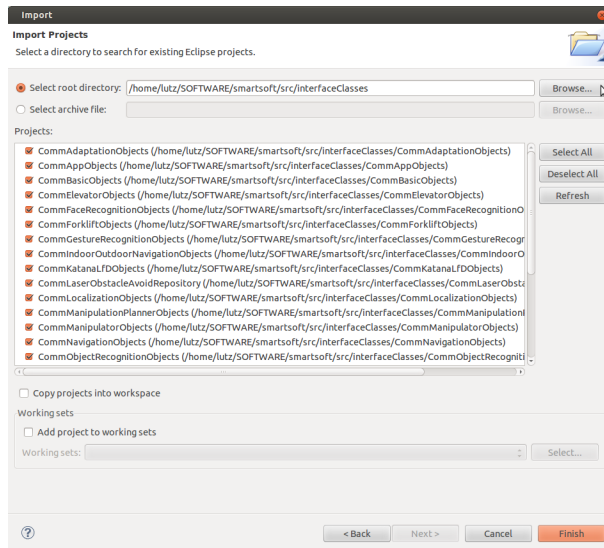


Figure 2.5. Import projects into the workspace, in this case communication/coordination repository projects projects.

Once imported the toolchain starts building the projects (reading files and generating code in case the models have changed). The progress can be seen at the bottom right task bar of the toolchain. Other activities such as the C++ indexer for example are also displayed there.

It is important to note that building in this context is always related to eclipse projects and the models, not to the generated results (e.g. C++ code) so no C++ compiler will be triggered.

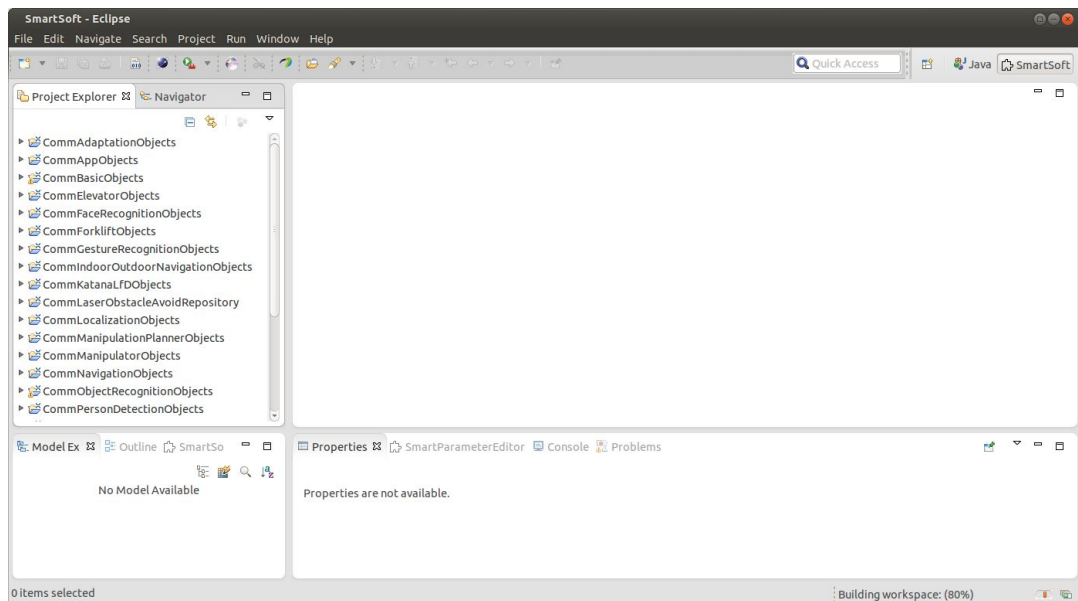


Figure 2.6. Toolchain with imported communication objects repository projects.

For components or deployment projects the same import procedure can be applied. How to use the imported projects and how to build new projects will be explained in the subsequent sections.

2.1.3.3. SmartMDSO Eclipse Projects

Communication objects, components and system configurations are Eclipse projects. The projects contain a model and source code which is generated from the model by a code generator. During code

generation, files for generated code and handwritten code are created (using the generation-gap pattern which links user-code and generated-code by inheritance). The src-gen folder contains the generated code. These files must not be changed by the user, because they are regenerated each time the code generator is started. Therefore, changes to these files will be lost. Files for handwritten code are located in the src-folder. These files are generated once and provides method skeletons for user implementation. However, this means that the files have to be adjusted/deleted manually after renaming/deleting the corresponding model element.

For automatic code generation and caching, make sure autobuild is activated in the workspace. For that, choose Project->Build Automatically in eclipse. If autobuild is enabled, a check mark is shown in front of the menu entry "Build Automatically".

2.1.3.4. Version Control Considerations

All folders and files of the projects except the folders "bin" and "build" should be committed. These folders contain files which are altered as soon as the project is opened in the toolchain. Therefore, it is not advisable to add these folders to the version control.

2.2. System Design View

The system design view is used to prepare the communication objects that components use for interacting with others through services. Communication objects are grouped to communication object repositories, that name a group of communication objects. To create a new communication/coordination repository project, select "File->New->SmartSoft Communication/Coordination Repository" (cf. figure 2.7), enter a name and choose a storage location. Typically the project is stored at "\$SMART_ROOT_ACE/src/interfaceClasses" and its name starts with "Comm" followed by the name of the repository.

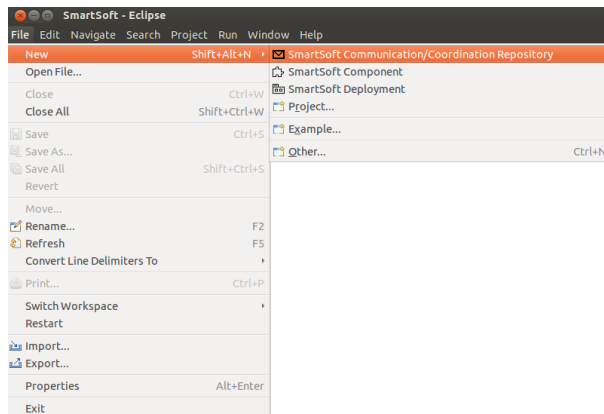


Figure 2.7. Create a new Communication/Coordination Project

Existing SmartSoft communication/coordination projects can be imported by selecting "File->Import". In the appearing dialog choose "General->Existing Projects into Workspace". See section "Starting for the first time" for more information.

The created project contains two different models. A communication objects model (file *.comm) and a Parameters model (file *.pardef). These models are located in the model-folder of the project.

2.2.1. Project Structure

Communication/coordination projects contain various folders and files. For the development of communication objects and parameters the following are important:

- **build:** Directory for compiling source code using cmake. This directory should not be versioned.
- **bin:** Temporary, toolchain internal folder. This directory should not be versioned.
- **model:** The model folder contains the textual models of the communication objects and parameters.
- **src:** After the code generation this folder contains user-specific source code for custom user-functions. For advanced users only (e.g. if you want to implement custom and complex access methods).
- **src-gen:** The src-gen folder contains source-code that is generated by the SmartSoftMDSD toolchain. Please do not modify these files, they will be overwritten each time the toolchain generator is run.
- **CMakeLists.txt:** This file is used to change the default cmake behavior or to add additional libraries and dependencies. Further information is provided in section 2.3.4.

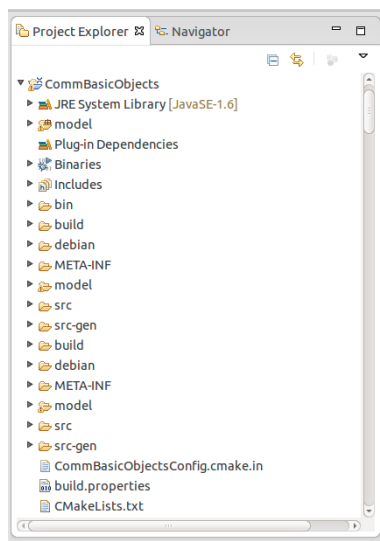


Figure 2.8. Structure of communication/coordination projects

2.2.2. Communication Objects

Communication objects parameterize the communication patterns and are transmitted by value. The following sections describes how communication objects are modeled and implemented. Additionally, a screencast which demonstrates the modeling of communication objects is available in section 3.1.1.

2.2.2.1. Modeling Communication Objects

Communication objects are modeled in a SmartSoft communication/coordination repository project. The model has the file extension '*.comm' and is located in the folder model/commObject. An empty model is generated once, after you created the project.

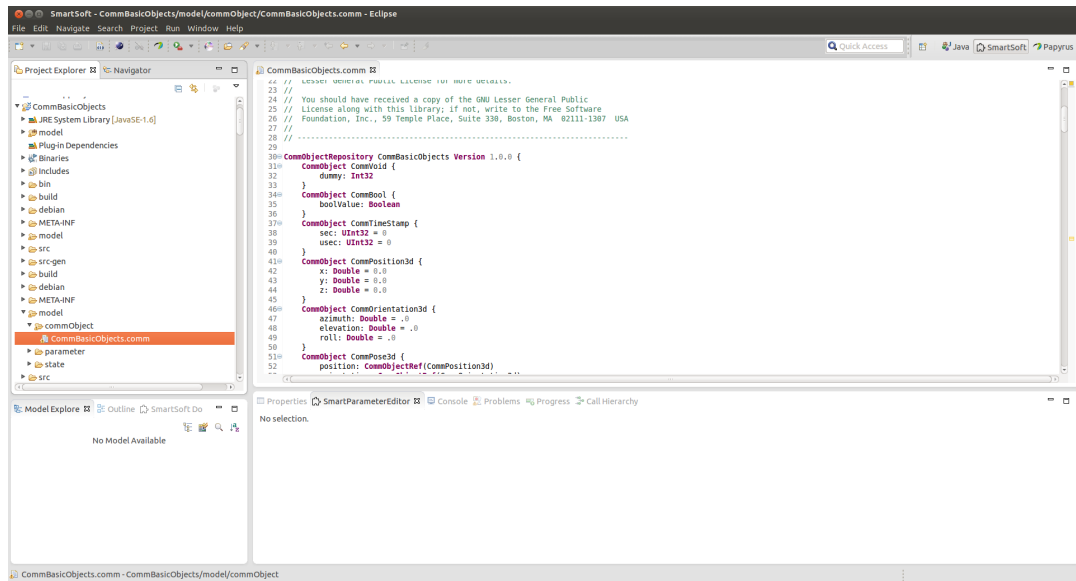


Figure 2.9. Model Communication Objects

The model consists of exactly one `CommObjectRepository` whose name must match the name of the project name. The `CommObjectRepository` of the project *CommBasicObjects* for example has to be defined as follows:

```
CommObjectRepository CommBasicObjects Version 1.0.0 {
}

```

Inside the `CommObjectRepository` an arbitrary number of communication objects, structs and enumerations can be defined.

A communication object is modeled as follows:

```
CommObject <name> {
  <name> : <data type> = <value>
}

```

The keyword `CommObject` is used to define a communication object. After the keyword the name of the communication object is given. This name should start with a capital letter. The elements of the communication objects are enclosed by curly braces. They consist of a name which should start with a small letter and a data type which can be optionally assigned with a default value. If no default value should be assigned, the assignment "`= <value>`" must be omitted. Possible data types are:

- Boolean
- CommObjectRef
- Double
- EnumRef
- Float
- Int8, Int16, Int32, Int64

- String
- StructRef
- UInt8, UInt16, UInt32, UInt64

Furthermore it is possible to use lists of these data types. In order to do so, an opening and closing square bracket has to be written behind the data type. The square brackets enclose the number of elements of the list. If the size of the list should be variable the symbol '*' is used.

If, for example, the velocity and rotation angle of a robot should be transmitted, the corresponding communication object can be defined as follows:

```
CommObject CommNavigationVelocity {  
    vX: Double = .0  
    vY: Double = .0  
    omega: Double = .0  
}
```

This example can be found in the *CommBasicObjects* repository. The name of the communication object is *CommNavigationVelocity* and it contains the attributes *vX*, *vY* and *omega*. All these attributes are of the data type *Double* and have the default value 0.0. The attribute *vX* specifies the velocity towards x and the attribute *vY* specifies the velocity towards y. The attribute *omega* is used to specify the rotation angle of the robot.

Lists are defined as follows:

```
CommObject CommPersonDetectionEventResult {  
    environment_id: UInt32= 0  
    person_id: UInt32[*]  
}
```

In this example the attribute *person_id* of the communication object *CommPersonDetectionEventResult* is a list of any number of *UInt32* values.

Additionally to the simple data types it is possible to use *EnumRef*, *StructRef* and *CommObjectRef*. These are references to Enums, Structs or communication objects. To be able to use such a data type the referenced Enum, Struct or communication object has to be defined inside the model. If an Enum, Struct or communication object of another SmartSoft communication/coordination repository should be referenced, the repository has to be imported by using the keyword 'ImportUri'. To import for example the *CommBasicObjects* repository the following line has to be added at the beginning of the model:

```
ImportUri "platform:/resource/CommBasicObjects/model/commObject/  
CommBasicObjects.comm"
```

Structs are defined with the keyword 'Struct'. After the keyword the name of the Struct is given. The attributes of the Struct are enclosed by curly braces. They consist of a name which should start with a small letter and a data type. Optionally a value can be assigned to the attribute. If no default value should be assigned, the assignment "= <value>" must be omitted. The data types are the same as in the communication object except of the data type *CommObjectRef*. This data type should not be used in Structs.

```
Struct <name> {
```

```
<name> : <data type> = <value>
}
```

Enumerations are defined with the keyword 'Enum'. After the keyword the name of the enumeration is given. The elements of the enumeration are enclosed by curly braces and consist of a name.

```
Enum <name> {
  <name>
}
```

An attribute which references an Enum can be defined as follows:

```
Enum ComparisonState {
  UNKNOWN
  GREATER
  LOWER
  INBETWEEN
}

CommObject CommBatteryEvent {
  chargeValue: Double = .0
  state: EnumRef(ComparisonState)
}
```

This example can be found in the CommBasicObject repository. The enumeration contains four items which specify the used comparison state. Inside the communication object CommBatteryEvent this enumeration is referenced. This means that the attribute *state* is of the data type ComparisonState. For referencing, it makes no difference whether the enumeration is defined before or after the communication object.

An example for referencing communication objects can be found in the CommBasicObjects repository:

```
CommObject CommTimeStamp {
  sec: UInt32 = 0
  usec: UInt32 = 0
}

CommObject CommDataFile {
  filename : String
  filesize : UInt32
  timestamp : CommObjectRef(CommTimeStamp)
  data : Int8[*]
  valid : Boolean
}
```

The communication object *CommDataFile* contains an attribute *timestamp* which is of the data type CommTimeStamp.

2.2.2.2. Generated files

The code generator will start as soon as the model of the SmartSoft communication/coordination repository is saved. The code generator creates C++ code for all communication objects of the repository. If

a communication object is deleted from the model, the corresponding files inside the src-folder have to be deleted manually. Otherwise the repository will not compile.

The following files are generated into the src-folder:

- <commObj name>.cc
- <commObj name>.hh

Standard getter and setter methods are generated automatically. If you need more advanced access methods, you can proceed as follows: In the <commObj name>.cc and <commObj name>.hh files methods of the communication object can be adjusted or added. For example the getter and setter methods of the communication object `CommNavigationVelocity` were adjusted as follows (`CommBasicObjects/src/CommBasicObjects/CommNavigationVelocity.hh`):

```
inline double get_vX(const double unit = 0.001) const { return
getVX() * (0.001 / unit); }
inline double get_vY(const double unit = 0.001) const { return
getVY() * (0.001 / unit); }

inline void set_vX(double v, const double unit = 0.001) { setVX( v
* (1000 * unit) ); }
inline void set_vY(double v, const double unit = 0.001) { setVY( v
* (1000 * unit) ); }
```

These methods are used to return and set the translation velocity in various units. The default unit is millimeters per second.

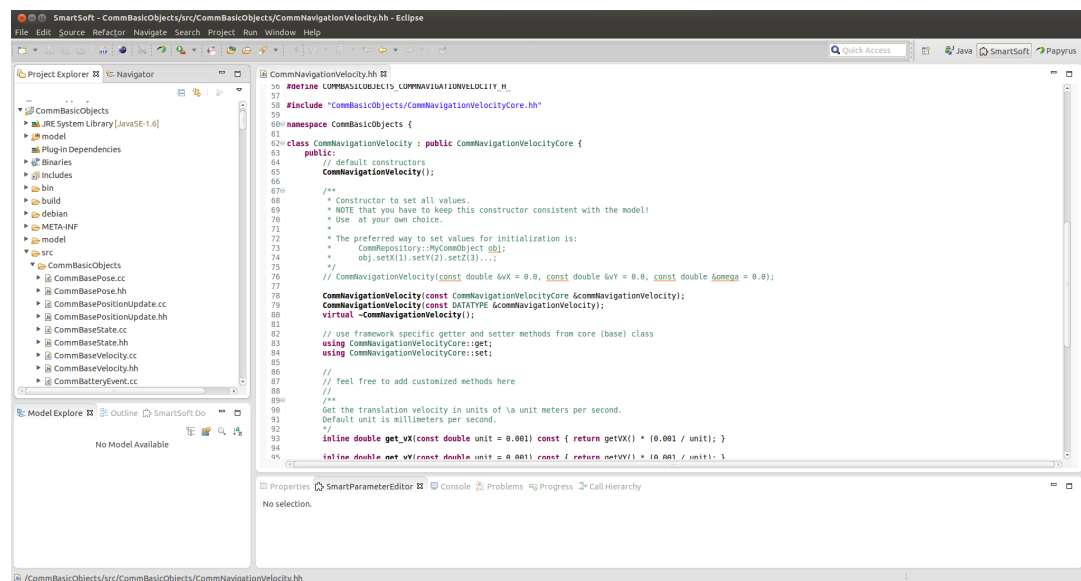


Figure 2.10. Add custom functions

2.2.3. Parameter Sets

In addition to the following description a screencast of modeling Parameter sets is shown in section 3.1.2.

Parameter sets are modeled in a SmartSoft communication/coordination repository project. The model has the file extension '*.pardef' and is located in the folder model/parameter. It consists of exactly one `ParamRepository` which contains an arbitrary number of parameter sets.

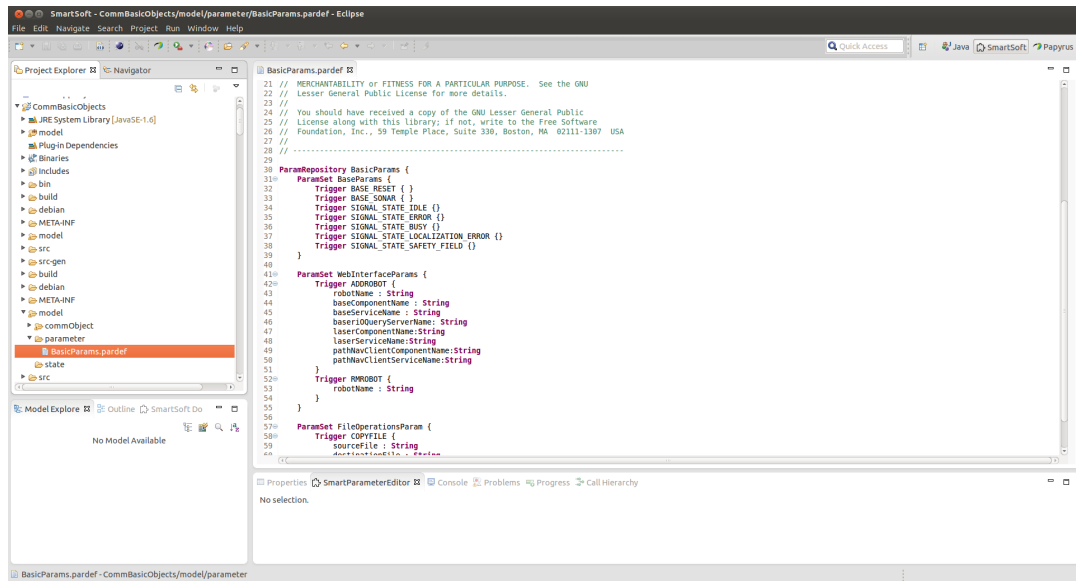


Figure 2.11. Model Parameter Sets

The ParamRepository of the project CommNavigationObjects for example is defined as follows:

```
ParamRepository CommNavigationObjects {
}
```

The definition of a parameter set starts with the keyword 'ParamSet' and is followed by a name.

```
ParamSet <name> {
}
```

Inside the parameter set parameters and trigger can be defined. These are enclosed by curly braces.

A parameter is defined as follows:

```
Param <name> {
  <name> : <data type>
}
```

The keyword 'Param' is used to define a parameter. After the keyword the name of the parameter is given. The elements of the parameter are enclosed by curly braces and consist of a name and a data type. Possible data types are:

- Boolean
- Double
- Enum
- Float

- Int8, Int16, Int32, Int64
- String
- UInt8, UInt16, UInt32, UInt64

Furthermore it is possible to use lists of these data types. In order to do so, an opening and closing square bracket has to be written behind the data type. The square brackets enclose the number of elements of the list. If the size of the list should be variable the symbol '*' is used.

A Trigger is defined as follows:

```
Trigger <name> {  
  <name> : <data type>  
}
```

Trigger are defined with the keyword 'Trigger'. After the keyword the name of the trigger is given. The elements of the trigger are enclosed by curly braces and consist of a name and a data type. The data types are the same as in the parameter definition.

The following example of a parameter set definition can be found in the CommNavigationObjects repository.

```
ParamSet MapperParams {  
  Trigger CURPARAMETER {  
    xsize : Int32  
    ysize : Int32  
    xpos  : Int32  
    ypos  : Int32  
    id    : Int32  
  }  
  
  Param CURLTM {  
    preoccupation : Enum { DISABLE ENABLE }  
    threshold     : Int32  
  }  
}
```

In this example the trigger CURPARAMETER is defined inside the parameter set MapperParams and contains the elements xsize, ysize, xpos, ypos and id. The parameter CURLTM contains the elements preoccupation and threshold.

2.2.4. Compile SmartSoft Communication/Coordination Repositories

To be able to use the the communication object during the component development they have to be compiled. To do so, right click on the repository and choose "Compile SmartMDS Project" (cf. figure 2.12).

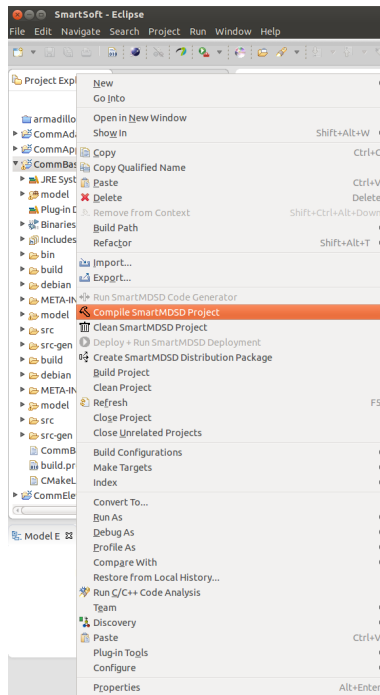


Figure 2.12. Compile a communication/coordination project

If the communication/coordination project has to be compiled without the toolchain, navigate to the repository and enter the following commands:

```
mkdir build
cd build
cmake ..
make
```

2.2.5. Documentation

For communication objects, structs and enumerations a *.dox file is generated automatically as soon as the model is saved. This file can be used to generate a documentation with doxygen. The documentation contains all communication objects, structs and enumerations of the CommObjectRepository with their names, attributes and a class reference. To add further information to the class reference the '@doc' keyword followed by a String has to be written on top of the communication object, struct or enumeration. The information will then be included in the documentation.

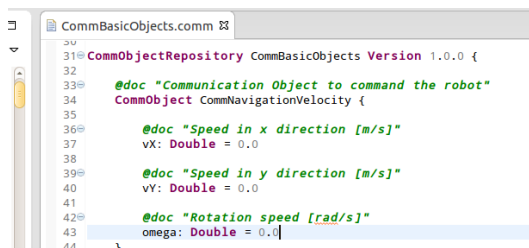


Figure 2.13. Documentation of communication objects

2.3. Component Development View

The component development is also shown in a screencast (section 3.1.3).

For the component development a new SmartSoft component project has to be created. To do so, choose "File->New->SmartSoft Component" (cf. figure 2.14), enter a name and choose a storage location. Typically the project is stored at "\$SMART_ROOT_ACE/src/components" and its name starts with "Smart" followed by the purpose of the component.

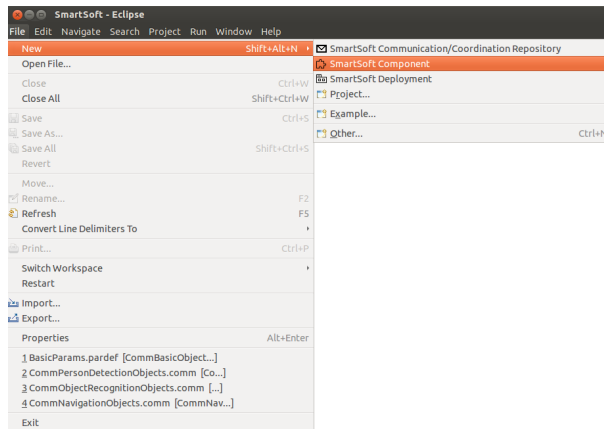


Figure 2.14. Create a new component project

Existing SmartSoft component projects can be imported by selecting "File->Import". In the appearing dialog choose "General->Existing Projects into Workspace".

The development of SmartSoft components consists of two steps. The component modeling and the component implementation. During the component modeling the external structure of the component is defined. The services which are used to communicate with other SmartSoft components, as well as the parameters which can be adjusted during the system composition are defined during this step. After the external structure is modeled, the code generator has to be started manually. The generated C++ files are used to implement the functionality of the component.

2.3.1. Component Projects

Component projects have a common folder structure as explained in the following:

- **build:** This directory contains the cmake generated files and should **not** be versioned.
- **bin:** Temporary, toolchain internal folder. This directory should **not** be versioned.
- **model:** The model folder contains the graphical model of the Component (*.di) and the human readable documentation model of the component (<component name>_documentation.compdoc).
- **src:** This folder contains of both, the initially generated C++ files and the custom C++ files. The generated C++ files in this folder will **not** be overwritten during the successive code egenerator calls and thus can be safely modified according to the component's specific needs.
- **src-gen:** Source-code that is generated by the SmartSoftMDSD toolchain. Please do not modify these files, they will be overwritten each time the toolchain generator is executed.
- **CMakeLists.txt:** This file is used to include additional sources for compilation and add external libraries or further smartsoft utilities and/or communication-objects. (Further information is provided in section 2.3.4)

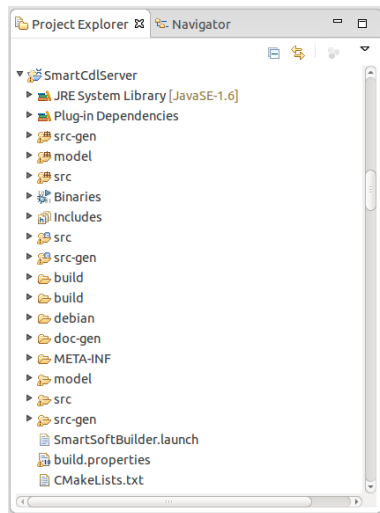


Figure 2.15. Structure of Component Projects

2.3.2. Component Modeling

The Component model is part of a SmartSoft Component project and is located in the model-folder.

2.3.2.1. Component Hull

The component hull is modeled graphically. The model consists of a component which provides and requires services in order to interact with other components and contains tasks, timers and parameters to implement a specific functionality. The various elements of the component are displayed in the tool palette of the SmartMDS Toolchain. In order to add an element to the component, the element has to be selected in the tool palette and then to be added to the component-diagram by clicking into it. The properties of the various elements can be adjusted in the properties tab if the element is selected in the model. To delete a model element right click on the element and choose "Delete Selected Element". Do not use the delete key on your keyboard to delete model elements. Due to a double assignment of the delete key the elements may be hidden instead of deleted. Figure 2.16 illustrates the modeling of the component hull with the SmartMDS Toolchain.

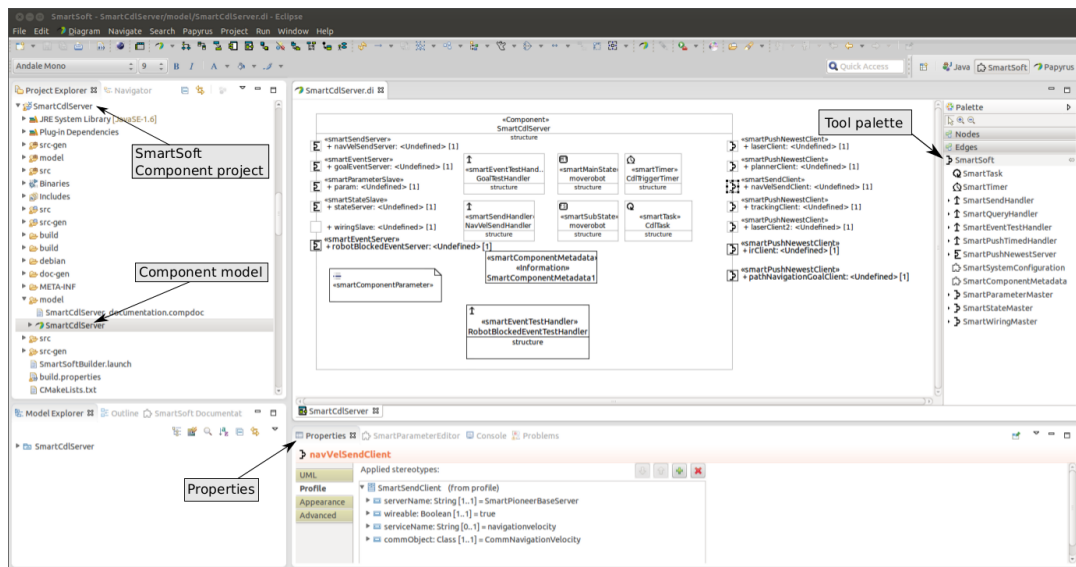


Figure 2.16. Modeling Component Hull

2.3.2.1.1. Communication Patterns

Communication patterns define how SmartSoft components communicate with each other. They consist of two complementary parts named service requestor and service provider representing a client/server (e.g. PushNewestClient/PushNewestServer), or, master/slave (e.g. StateMaster/StateSlave) relationship. The various communication patterns are described in section 1.1.3.2. To model a service requestor or service provider add a communication port and, where necessary, a handler to the model. To do so, select the element in the tool palette and click on the component. Afterwards, adjust the properties of the port or handler in the properties tab. The following list presents the individual properties:

event:

- SmartEventServer:
 - *eventState*: Defines the communication object of the event state, which is used by the SmartEventTestHandler (see below) to check the event parameter.
 - *smartEventTestHandler*: Defines the SmartEventTestHandler which is used to check whether an event fires according to the client-specific eventParameter (see next).
 - *eventParameter*: A client registers to an EventServer with an event activation parameter. The eventParameter defines the communication object of this event activation parameter to be used within the SmartEventTestHandler (above). The communication object must match the communication object of the eventParameter of the SmartEventClient (see below).
 - *eventResult*: Specifies the communication object of the event result, which is the actually communicated event with according data-payload.
- SmartEventClient:
 - *smartEventHandler*: Defines the SmartEventHandler which is used as a callback-handler to receive the event results.
 - *serverName*: Specifies the name of the component that provides the event result. (This value is refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *wireable*: Defines whether or not the port can be wired dynamically (using the Wiring pattern).
 - *serviceName*: Specifies the name of the SmartEventServer. (This value is also refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *eventParameter*: Defines the communication object of the event parameter. The client registers with an event parameter to the SmartEventServer in order to parametrise the server-side event-filter (implemented inside of the according SmartEventTestHandler in the server). This allows to specify events of interest and reduces unnecessary communication overhead.
 - *eventResult*: Specifies the communication object of the event result.

push timed:

- SmartPushTimedHandler: is a call-back handler used by SmartPushTimedServer to periodically publish updates.
 - *isActive*: Specifies whether the SmartPushTimedHandler is active or passive. If the handler is set passive, the processing of incoming requests is driven by the upcalling thread of the SmartPushTimedServer. A passive handler should avoid using blocking system-calls (such as e.g. sleeps, waits, semaphores, condition variables, etc.).
- SmartPushTimedServer:
 - *timeUnit*: Specifies the time unit of the cycle decorator (s, ms, us or ns).

- *cycle*: Specifies the time interval at which new data is provided.
- *smartPushTimedHandler*: Specifies the SmartPushTimedHandler which is used to periodically send data.
- *commObject*: Specifies the communication object which is send to a client.
- SmartPushTimedClient:
 - *interval*: It is possible to get every n-th update to reduce unnecessary communication overhead. The prescaler of the server cycle is specified with this parameter.
 - *serverName*: Specifies the name of the component which provides the data. (This value is refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *wireable*: Defines whether or not the port can be wired dynamically (using the wiring pattern).
 - *serviceName*: Specifies the name of the SmartPushTimedServer. (This value is also refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *commObject*: Specifies the received Communication Object. The communication object must match the communication object which is send via the SmartPushTimedServer.

push newest:

- SmartPushNewestServer:
 - *commObject*: Specifies the communication object which is published to all subscribed clients.
- SmartPushNewestClient:
 - *serverName*: Specifies the name of the component which provides the data. (This value is refined during System Configuration (see section 2.4.2) and can therefore be left blank.)
 - *wireable*: Defines whether or not the port can be wired dynamically (using the Wiring pattern).
 - *serviceName*: Specifies the name of the SmartPushNewestServer. (This value is also refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *commObject*: Specifies the received communication object. The communication object must match the communication object which is published from the SmartPushNewestServer.

query:

- SmartQueryHandler: is a callback handler used by SmartQueryServer to process incoming queries.
 - *isActive*: Specifies whether the SmartQueryHandler is active or passive. If the handler is set passive, the processing of incoming requests is driven by the upcalling thread of the SmartQueryServer and thus should avoid any blocking calls.
- SmartQueryServer:
 - *smartQueryHandler*: Specifies the upcall-handler reference (SmartQueryHandler, see above) to process incoming query requests.
 - *commRequestObject*: Specifies the request communication object.
 - *commAnswerObject*: Specifies the response communication object.
- SmartQueryClient:

- *serverName*: Specifies the name of the component which provides the according QueryServer. (This value is refined during system configuration (see section 2.4.2) and can therefore be left blank.)
- *wireable*: Defines whether or not the port can be wired dynamically (using the Wiring pattern).
- *serviceName*: Specifies the name of the SmartQueryServer. (This value is also refined during System Configuration (see section 2.4.2) and can therefore be left blank.)
- *commRequestObject*: Specifies the communication object of the request.
- *commAnswerObject*: Specifies the communication object of the response.

send:

- SmartSendHandler: is a callback handler used by SmartSendServer to process incoming sends.
- *isActive*: Specifies whether the SmartSendHandler is active or passive. If the handler is set passive, the processing of incoming requests is driven by the upcalling thread of the SmartSendServer and thus should avoid any blocking calls.
- SmartSendServer:
 - *smartSendHandler*: Specifies the smartSendHandler which implements the callback to receive and to process incoming data.
 - *commObject*: Specifies the communication object of the received data.
- SmartSendClient:
 - *serverName*: Specifies the name of the component which receives the send data. (This value is refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *wireable*: Defines whether or not the port can be wired dynamically (using the Wiring pattern).
 - *serviceName*: Specifies the name of the SmartSendServer. (This value is also refined during system configuration (see section 2.4.2) and can therefore be left blank.)
 - *commObject*: Specifies the communication object which is send.

Before a communication object can be assigned to a communication port, it needs to be imported (if not already done). Therefore, right click anywhere inside of the component diagram and select "Import Communication/Coordination Objects". In the next dialog, select the required communication objects and press the OK Button. After the communication object was imported, select the communication port and add the communication object (as shown in figure figure 2.17).

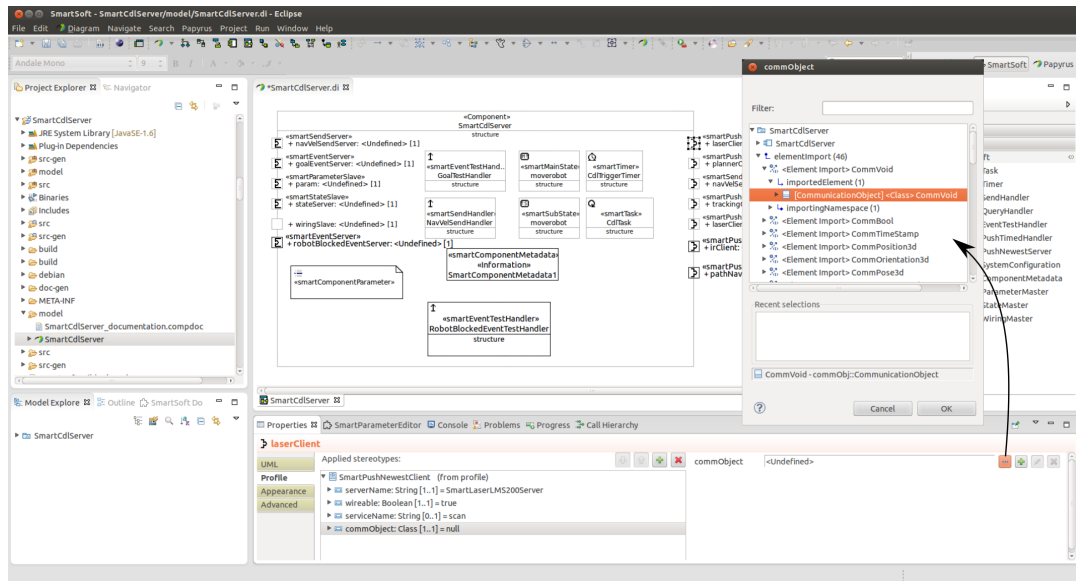


Figure 2.17. Select Communication object

In order to assign a handler to a communication port, the corresponding handler has to be modeled in the component model first (using the tool palette). After that, select the handler parameter in the communication port and press the "..."-Button on the right. In the next dialog select the handler (<component name> -> <component name> -> nestedClassifier -> <handler>) and press the OK button (cf. figure 2.18).

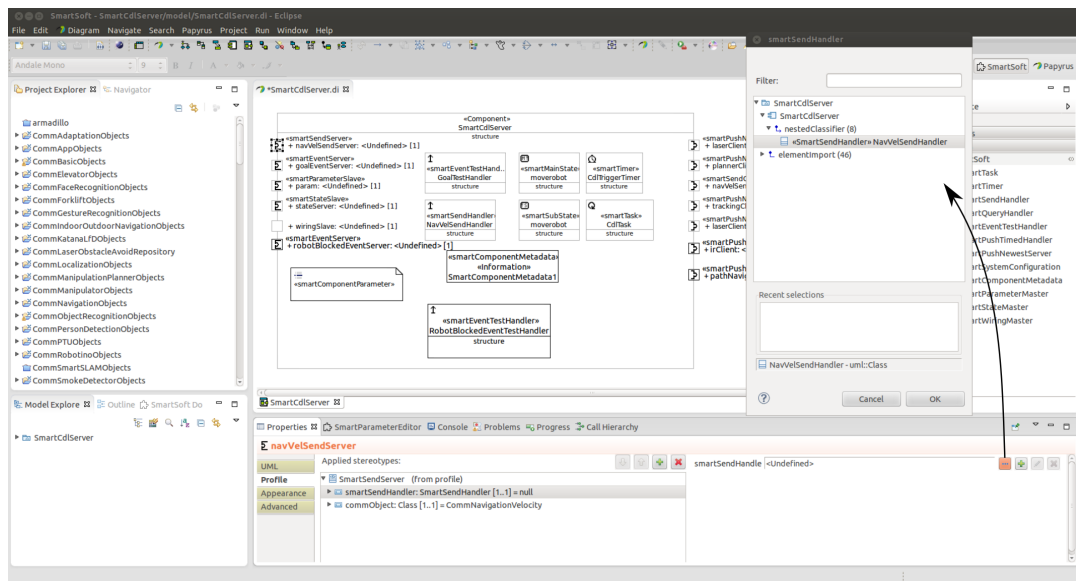


Figure 2.18. Select handler

2.3.2.1.2. SmartTask

The following settings can be made:

- *timeUnit*: Specifies the time unit of the period decorator (s, ms, us or ns).
- *isPeriodic*: Specifies whether or not the Task is called periodic in a specified time interval.
- *period*: Specifies the time interval at which the task is called.

2.3.2.1.3. SmartComponentMetadata

The SmartComponentMetadata has to be added to every component. It contains version and dependency information.

2.3.2.2. Component Parameters

There are two main types of parameters. There are parameters to configure a component initially at system-configuration / deployment time but not during runtime. Then there are parameters that can be used to configure a component initially and at run-time.

Parameters can be defined within the component (component internal) and outside of the component (component external) for reusing the definitions in multiple components. Parameters that can be set at run-time can be defined with the keyword `ExtendedParam` within the component or reuse existing definitions with the keyword `ParamSetInstance`. Parameters that can only be set initially can be defined within the component using the keyword `InternalParam`. They cannot be defined outside of the component.

A component that can be configured at run-time needs a `SmartParameterSlave` service. A component configuring other components does this via the `SmartParameterMaster`.

2.3.2.2.1. SmartParameterMaster

Provides a generic port to set configurations of components at runtime. Typically the master part is used within the sequencer, which is in control and coordinating the system. In total one parameter master can send parameter-sets to several Parameter Slaves. There are three different ways to use the parameter to configure other components (which are encoded in the `CommParameterRequest`):

- Simple parameters such as `SETPOSE(x,y) CHANGEMAP("MapName"), ...` - these are name-value configurations
- Triggers - used to trigger/start actions or activities
- COMMIT - a special kind of trigger that tells a component that a sequence of configurations is complete and the component can from now on use the consistent set of new parameters.

2.3.2.2.2. SmartParameterSlave

The slave part of the parameter is used when your component is subject to configuration by other components at run-time. The user needs at most one instance of the parameter slave per component.

2.3.2.2.3. SmartComponentParameter

The `SmartComponentParameter` is used to model the parameters. The elements are defined in a `parameterDefinition`. The `parameterDefinition` can be modified with the `SmartParameterEditor` which is to the right of the properties tab. To open the `parameterDefinition` in the editor click on the `SmartParameterEditor` tab and then on the `SmartComponentParameter`.

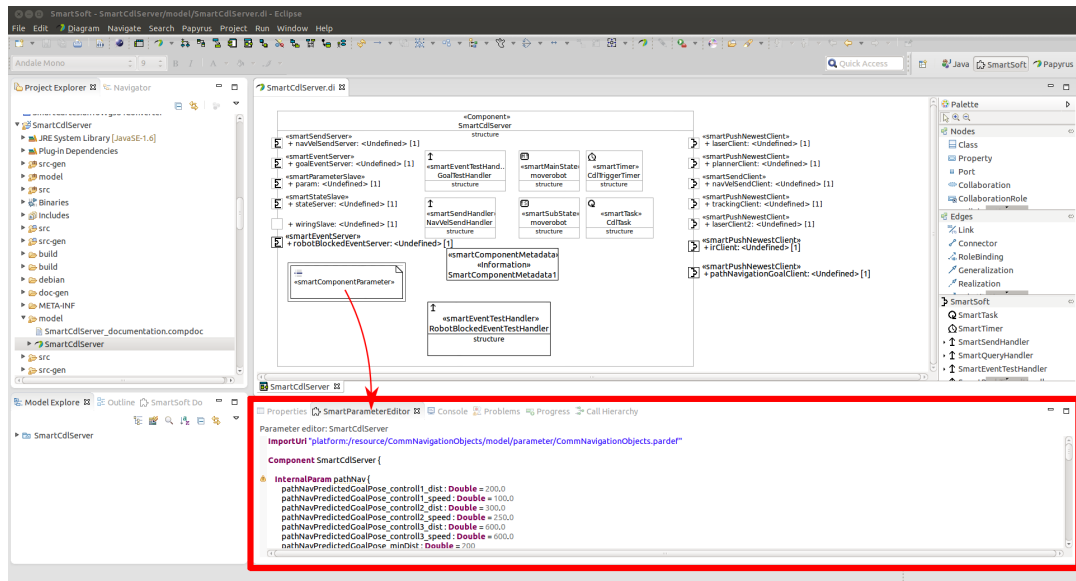


Figure 2.19. Model Component parameter

The parameters are modeled as follows:

```
Component <name> {
  <parameter>
}
```

At first the component of the SmartComponentParameter is defined. The name must match the name of the component in which the SmartComponentParameter is modeled. Inside the component internal parameters, extended parameters, extended trigger and parameter set instantiations are defined.

Internal parameters are modeled as follows:

```
InternalParam <name> {
  <name> : <data type> = <value>
}
```

The keyword 'InternalParam' is used to define an internal parameter. After the keyword the name of the internal parameter is given. This name should start with a capital letter. The attributes of the internal parameter are enclosed by curly braces. They consist of a name and a data type. The attributes must be assigned with a default value. Possible data types are:

- Boolean
- Double
- Enum
- Float
- Int8, Int16, Int32, Int64
- String
- UInt8, UInt16, UInt32, UInt64

Additionally, lists of these data types can be used.

If, for example, the maximum velocity and steering should be defined with parameters, the internal parameter can be defined as follows:

```
InternalParam settings{
  max_velocity : Double = 1.0
  max_steering : Double = 1.2
}
```

Extended parameters are modeled as follows:

```
ExtendedParam <name> {
  <name> : <data type> = <value>
}
```

Extended parameters are defined with the keyword 'ExtendedParam'. After the keyword the name of the extended parameter is given. The name should start with a capital letter. The attributes of the extended parameter are enclosed by curly braces. They consist of a name, a data type and a default value. The possible data types are similar to the data types of internal parameters.

Extended trigger are modeled as follows:

```
ExtendedTrigger <name> (active|passive) {
  <name> : <data type>
}
```

The keyword 'ExtendedTrigger' is used to define extended trigger. After the keyword the name of the extended trigger followed by the keyword 'active' or 'passive' is given. The attributes of the extended trigger are enclosed by curly braces. They consist of a name and a data type. The data types are similar to the data types of internal parameters.

As mentioned before, parameters can be defined outside of the component and can be reused. For that purpose the model of the parameter set has to be imported first:

```
ImportUri <path>
```

The parameter set is then instantiated as follows:

```
ParamSetInstance <repository>.<param set name> {
  InstantiateTrigger <trigger name> (active|passive)
  InstantiateParam <parameter name> {
    this.<attribute name> = <value>
  }
}
```

The instantiation of a parameter set is defined with the keyword 'ParamSetInstance' followed by the repository and name of the imported parameter set. The parameters and trigger which should be instantiated are enclosed by curly braces. Trigger are instantiated with the keyword 'InstantiateTrigger'

followed by the name of the trigger and either the keyword 'active' or 'passive'. Parameters are instantiated with the keyword 'InstantiateParam' followed by the name of the parameter and the attributes which are enclosed by curly braces. The attributes which should be assigned with a value are selected with the this-operator.

An example of the instantiation of a Parameter set can be found in the component *SmartCdlServer*:

```
ImportUri "platform:/resource/CommNavigationObjects/model/
parameter/CommNavigationObjects.pardef"

Component SmartCdlServer {
  ParamSetInstance CommNavigationObjects.CdlParameter {
    InstantiateTrigger SETSTRATEGY passive
    InstantiateParam PATHNAVFREEBEHAVIOR{this.free =
enum.DEACTIVATE}
  }
}
```

In this example the passive trigger SETSTRATEGY and the parameter PATHNAVFREEBEHAVIOR are instantiated.

Please refer to the chapter of component implementation and the tutorial to see how these parameters are used from within the component implementation.

2.3.2.2.4. Parameter Documentation

In addition attributes of parameters, extended parameters, extended trigger, instantiated parameters and instantiated triggers can be documented. This can be done in the SmartParameterEditor by writing the '@doc' keyword followed by a String on top of the attribute, parameter or trigger. The documentation of the attributes max_velocity and max_steering of the internal parameter settings for example can be added as follows. Use this within the parameter modeling.

```
// Parameter within the component
InternalParam settings{
  @doc"Defines the maximum velocity. This is the velocity [m/s]
which will be sent when the joystick axis is at full peak."
  max_velocity : Double = 1.0

  @doc"Defines the minimum steering angle. This is the angle [rad]
which will be sent as omega when the joystick axis is
at full peak."
  max_steering : Double = 1.2
}
```

2.3.3. Component Implementation

Before implementing the component, the code generator has to be started. To do so, right click on the component model and select "Run SmartMDS D Code Generator" (cf. figure 2.20).

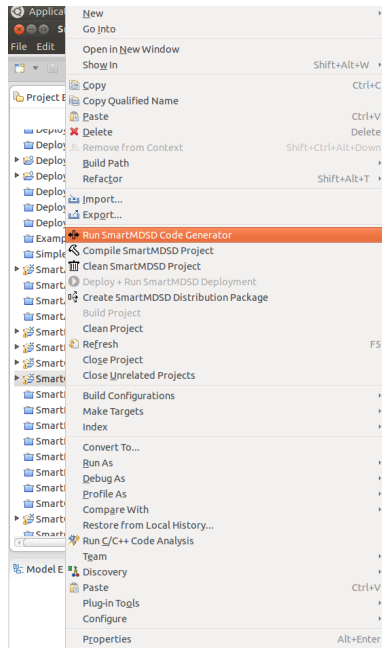


Figure 2.20. Code generation

2.3.3.1. Generated Files

Depending on the elements of the model, files are created during the code generation. For every **component** the files

- CompHandler.cc
- CompHandler.hh
- <component name>Core.cc
- <component name>Core.hh

are generated into the src-folder.

The *CompHandler.** files contain code which is executed after the component is started or terminated. By default all services, tasks and timer of the component are started in these files. Afterwards the component is notified that the setup/initialization is finished. The *<component name>Core.** files can be used to declare variables and methods which should be accessible from all other files inside the component.

For **SmartTasks** the following files are generated into the src-folder:

- <Task name>.cc
- <Task name>.hh

The *<Task name>.** files contain a constructor, destructor and the methods `on_entry()`, `on_execute()` and `on_exit()`. The method `on_entry()` is called once, each time the task is started and can be used to initialize procedures. The method `on_exit()` is called once at the end of the thread and is typically used to clean-up resources which were initialized in the `on_entry()` method. The method `on_execute()` is called periodically in the thread and contains the logic of the component.

For a **SmartEventClient** the following files are generated into the src-folder:

- <eventHandler name>.cc

- <eventHandler name>.hh

The <eventHandler name>.* files contain the method `handleEvent(const CHS::EventId id, const <Communication Object> &r)` which is called as soon as an event is received.

For a **SmartEventServer** the files

- <eventTestHandler name>.cc
- <eventTestHandler name>.hh

are generated into the src-folder. The <eventTestHandler name>.* files contain the method `testEvent(<Communication Object> &p, <Communication Object> &r, const <Communication Object> &s)` which is used to check whether the event condition is true and the event fires. The specific check has to be added by the user. Thereby the Communication object p is the event parameter, the communication object r is the event result and the communication object s is the event state. If the event condition is true the method must set the event result and return the value 'true'. In contrast, if the event condition is false the method must return the value 'false'.

For a **SmartSendServer** the following files are generated into the src-folder:

- <sendHandler name>.cc
- <sendHandler name>.hh

The <sendHandler name>.* files contain the method `handleSend(const <Communication Object> &r)` which is called as soon as a communication object is received.

For a **SmartQueryServer** the files

- <queryHandler name>.cc
- <queryHandler name>.hh

are generated into the src-folder. The <queryHandler name>.* files contain the method `handleQuery(CHS::QueryServer<<Communication Object>, <Communication Object>> &server, const CHS::QueryId id, const <Communication Object> &request)` which is called as soon as a request is received.

For a **SmartPushTimedServer** the following files are generated into the src-folder:

- <pushTimedHandler name>.cc
- <pushTimedHandler name>.hh

The <pushTimedHandler name>.* files contain the method `handlePushTimer(CHS::PushTimedServer<<Communication Object>> &server)` which is used to send data periodically.

For a **SmartStateSlave** the files

- SmartStateChangedHandler.cc
- SmartStateChangedHandler.hh

are generated into the src-folder. The *SmartStateChangedHandler*.* files contain the methods `handleEnterState(const std::string &substate)` which is called as soon as a substate is entered and `handleQuitState(const std::string &substate)` which is called as soon as a substate is left.

For a **SmartComponentParameter** the files

- ParameterStateStruct.cc
- ParameterStateStruct.hh

are generated into the `src`-folder. The `ParameterStateStruct.*` files contain the method `handleCOMMIT(const ParameterStateStruct &commitState)` which is used to implement consistency checks which ensure that the incoming parameter meets internal constraints.

2.3.3.2. Start Services and Tasks

Services and tasks are connected and started in the `CompHandler.cc` file. By default all services and tasks of the component are started in the method `onStartup()`. However, this code can be adapted to your needs. If, for example, a component has several services, but only the specific service "imuDataPushTimedClient" should be started the line

```
COMP->connectAndStartAllServices();
```

has to be replaced with

```
COMP->connectImuDataPushTimedClient(
    COMP->connections.imuDataPushTimedClient.serverName,
    COMP->connections.imuDataPushTimedClient.serviceName
);
```

Look at the generated implementation of `onStartup()` for more examples. Often the start of a service should be configurable with an internal parameter. To do so, surround the start of the service with an IF-statement, e.g. only call `COMP->connectImuDataPushTimedClient()` if a specific condition is true (cf. fig. 2.21).

The easiest way to change the code of the `CompHandler` is to copy the code of the default method which contains the startup code of all services and adjust this copied code to your needs. The default method can be found in the `<component name>.cc` file which is located in the `src-gen` folder.

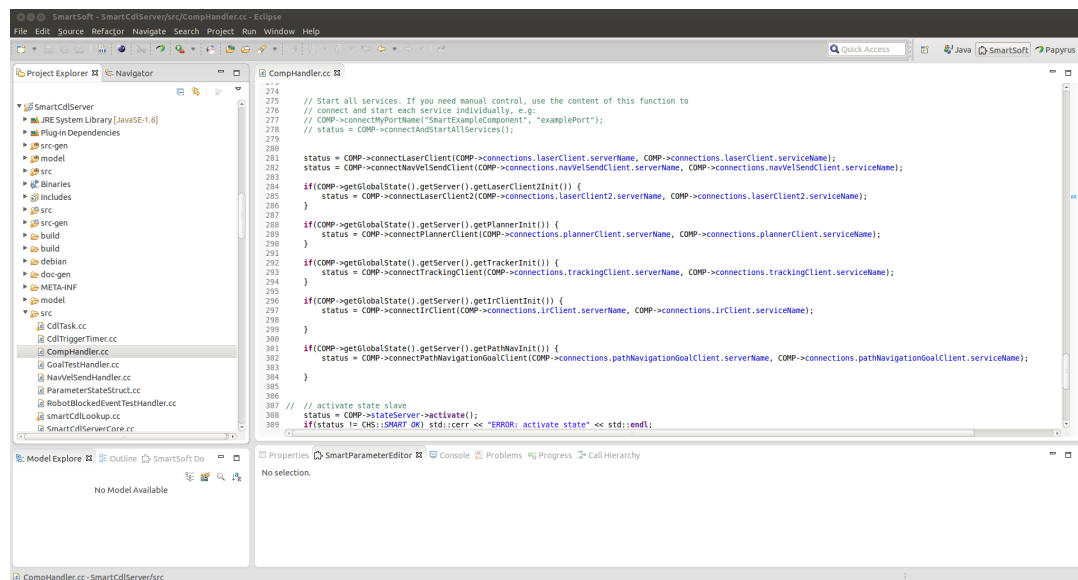


Figure 2.21. Start services

2.3.3.3. Using Communication Objects

To be able to use communication objects inside a C++ implementation the corresponding SmartSoft communication/coordination repository project has to be compiled (cf. section 2.2.4) and the `*.hh` file

of the communication object has to be included. If, for example, the communication object *CommNavigationVelocity* should be used in the C++ implementation of a component, it has to be included as follows:

```
#include "CommBasicObjects/CommNavigationVelocity.hh"
```

Now, the communication object can be instantiated:

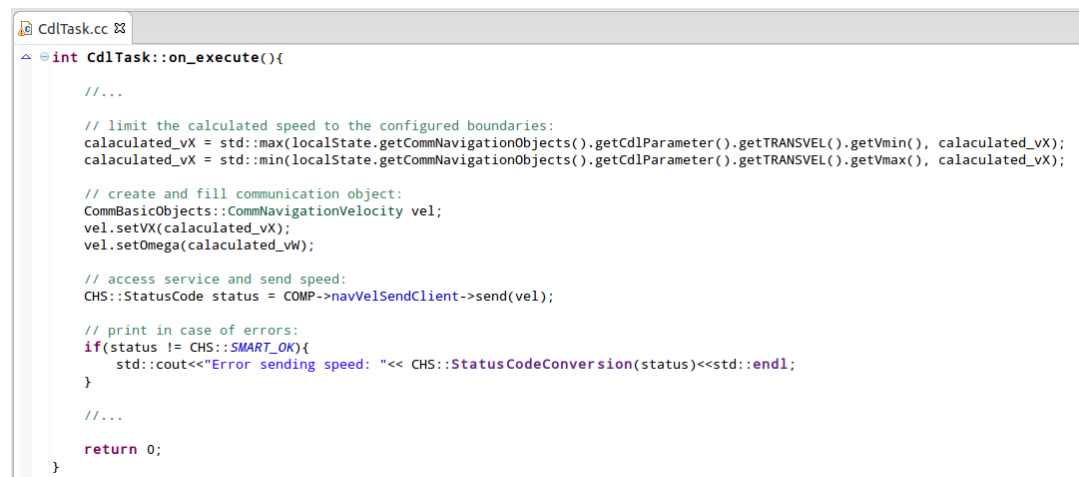
```
CommBasicObjects::CommNavigationVelocity vel;
```

The attributes of the communication object can be accessed via getter and setter methods:

```
vel.set_vX(x);
vel.set_omega(omega);
// using the builder pattern, set multiple values in one line:
vel.set_vX(x).set_omega(omega);
```

2.3.3.4. Using Services

There are different methods to exchange data for the different communication patterns. In the following the most important methods are presented. Further information can be found in the SmartSoft/ACE reference at <http://servicerobotik-ulm.de/drupal/doxygen/aceSmartSoft/>. For an example how to use a service from your implementation, see fig. 2.22



```

CdTask.cc
int CdTask::on_execute(){
    //...

    // limit the calculated speed to the configured boundaries:
    calculated_vX = std::max(localState.getCommNavigationObjects().getCdlParameter().getTRANSVEL().getVmin(), calculated_vX);
    calculated_vX = std::min(localState.getCommNavigationObjects().getCdlParameter().getTRANSVEL().getVmax(), calculated_vX);

    // create and fill communication object:
    CommBasicObjects::CommNavigationVelocity vel;
    vel.setVX(calculated_vX);
    vel.setOmega(calculated_vW);

    // access service and send speed:
    CHS::StatusCode status = COMP->navVelSendClient->send(vel);

    // print in case of errors:
    if(status != CHS::SMART_OK){
        std::cout<<"Error sending speed: "<< CHS::StatusCodeConversion(status)<<std::endl;
    }

    //...

    return 0;
}

```

Figure 2.22. Example implementation of a task using a service. [4]

event. The SmartEventClient subscribes for an event and receives the event result, which is provided by the SmartEventServer.

SmartEventClient:

- StatusCode activate(const EventMode mode, const P& parameter, EventId& id): *Activate an event with the provided parameters in either "single" or "continuous" mode.*
- StatusCode deactivate(const EventId id): *Deactivate the event with the specified identifier. An event must always be deactivated, even if it has already fired in single mode. This is just necessary for cleanup procedures and provides a uniform user API independently of the event mode. Calling deactivate() while there are blocking calls aborts them with the appropriate status code.*

- **StatusCode tryEvent(const EventId id):** *Check whether event has already fired and return immediately with status information. This method does not consume an available event.*
- **StatusCode getEvent(const EventId id, E& event):** *Blocking call which waits for the event to fire and then consumes the event. This method consumes an event. Returns immediately if an unconsumed event is available. Blocks otherwise till event becomes available. If method is called concurrently from several threads with the same id and method is blocking, then every call returns with the same event once the event fired. If there is however already an unconsumed event available, then only one out of the concurrent calls consumes the event and the other calls return with appropriate status codes.*
- **StatusCode getNextEvent(const EventId id, E& event):** *Blocking call which waits for the next event. This method waits for the next arriving event to make sure that only events arriving after entering the method are considered. Method consumes event. An old event that has been fired is ignored (in contrary to getEvent()). If method is called concurrently from several threads with the same id, then every call returns with the same event once the event fired.*

SmartEventServer:

- **StatusCode put(const S& state):** *Initiate testing the event conditions for the activations.*

pushNewest. The SmartPushNewestServer provides data for SmartPushNewestClients whenever new data is available.

SmartPushNewestServer:

- **StatusCode put(const T& d):** *Send updated data to all subscribed clients*

SmartPushNewestClient:

- **StatusCode getUpdate(T& d):** *Non-blocking call to immediately return the latest available data buffered at the client side from the most recent update. No data is returned as long as no update is received since subscription. To avoid returning old data, no data is returned after the client got unsubscribed.*
- **StatusCode getUpdateWait(T& d):** *Blocking call which waits until the next update is received. Blocking is aborted with the appropriate status if either the client gets unsubscribed or disconnected or if blocking is not allowed anymore at the client.*

pushTimed. The SmartPushNewestServer provides data in specified time intervals for SmartPushTimedClients.

SmartPushTimedServer:

- **StatusCode put(const T& d):** *Provide new data which is sent to all subscribed clients taking into account their individual update cycles. Update cycles are always whole-numbered multiples of the server update cycle.*

SmartPushTimedClient:

- **StatusCode getUpdate(T& d):** *Non-blocking call to immediately return the latest available data buffered at the client side from the most recent update. No data is returned as long as no update is received since subscription. To avoid returning old data, no data is returned after the client is unsubscribed or when the server is not active.*
- **StatusCode getUpdateWait(T& d):** *Blocking call which waits until the next update is received. Blocking is aborted with the appropriate status if either the server gets deactivated, the client gets unsubscribed or disconnected or if blocking is not allowed any more at the client.*

query. The SmartQueryClient sends a request containing individual parameters and receives an individual result from the SmartQueryServer.

SmartQueryClient:

- `StatusCode query(const R& request, A& answer)`: *Perform a blocking query and return only when the query answer is available. Member function is thread safe and thread reentrant.*
- `StatusCode queryRequest(const R& request, QueryId& id)`: *Perform a query and receive the answer later, returns immediately. Member function is thread safe and reentrant.*
- `StatusCode queryReceive(const QueryId id, A& answer)`: *Check if answer is available. Non-blocking call to fetch the answer belonging to the given identifier. Returns immediately. Member function is thread safe and reentrant.*
- `StatusCode queryReceiveWait(const QueryId id, A& answer)`: *Wait for reply. Blocking call to fetch the answer belonging to the given identifier. Waits until the answer is received.*
- `StatusCode queryDiscard(const QueryId id)`: *Discard the pending answer with the identifier id. Call this member function if you do not want to get the answer of a request anymore which was invoked by `queryRequest()`. This member function invalidates the identifier id.*

`SmartQueryServer`:

- `StatusCode answer(const QueryId id, const A& answer)`: *Provide answer to be sent back to the requestor. Member function is thread safe and thread reentrant.*

send. The `SmartSendClient` provides data which is received by a `SmartSendServer`.

`SmartSendClient`:

- `StatusCode send(const C& c)`: *Perform a one-way communication. Appropriate status codes make sure that the information has been transferred.*

state and parameter. For descriptions on state and parameter, please refer to the doxygen reference of `SmartSoft/ACE`. For an example how to use the parameter, please refer to the tutorial within this handbook.

2.3.3.5. Status Codes

Status codes are used for the status and error handling when using `SmartSoft` methods, especially when interacting with services. To convert a status code into readable ASCII representation the following method can be used:

```
std::string StatusCodeConversion(StatusCode code)
```

The method can be used as follows:

```
CHS::StatusCode status = COMP->connectAndStartAllServices();
if(status != CHS::SMART_OK) {
    std::cout << "Error connecting services: " <<
    CHS::StatusCodeConversion(status);
}
```

2.3.3.6. Component Wide Variables

Variables which should be accessible from all files of the component project can be declared in the `<component name>Core.hh` file. To prevent the access of several threads at the same time a `Mutex` should be added to this variables. The variable can be accessed as follows:

```
COMP-><variable name>
```

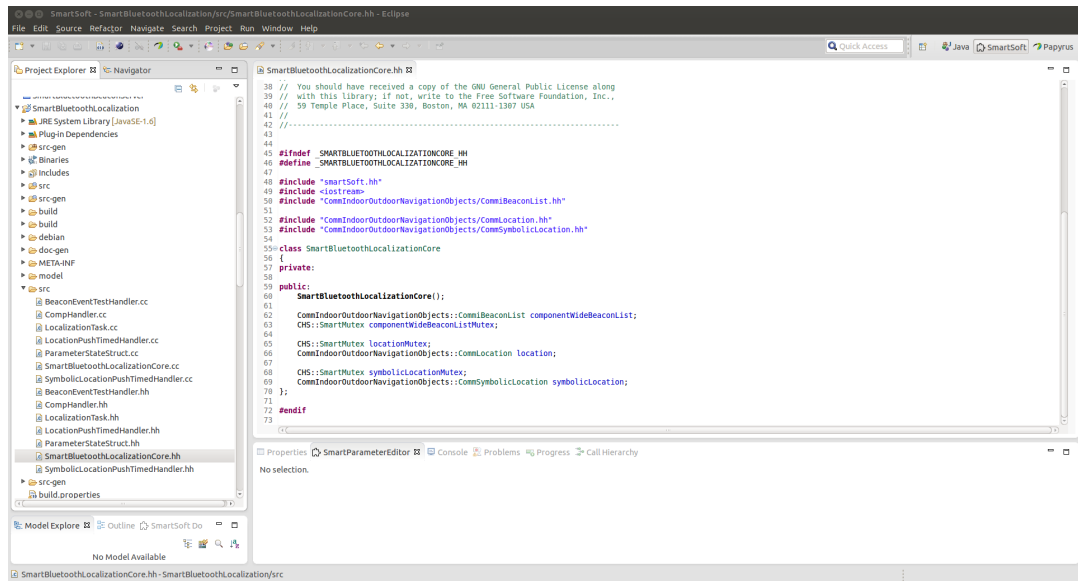


Figure 2.23. Add Component wide variables

A SmartMutex can be used to prevent simultaneous access of different threads. An example of a mutex for a global variable could be found in the component *SmartBluetoothLocalization* (SmartBluetoothLocalizationCore.hh):

```
CommIndoorOutdoorNavigationObjects::CommBeaconList
componentWideBeaconList;
CHS::SmartMutex componentWideBeaconListMutex;
```

To acquire or release the lock ownership of the component wide variable componentWideBeaconList, the following code has to be used:

```
COMP->componentWideBeaconListMutex.acquire();
COMP->componentWideBeaconList.clearBeaconList();
COMP->componentWideBeaconListMutex.release();
```

2.3.3.7. Using Parameters Within the Component

To access an attribute of a parameter within the component the following code is used:

```
COMP->getGlobalState().get<parameter name>().get<attribute
name>();
```

The settings.max_velocity, for example, is accessed as follows:

```
COMP->getGlobalState().getSettings().getMax_velocity();
```

In case these parameters change at runtime, we strongly recommend to work on a copy of the global state:

```
ParameterStateStruct localState = COMP->getGlobalState();
```

```
localState.getSettings().getMax_velocity();
```

2.3.4. Compile SmartSoft Component Projects

To be able to use the developed components during system composition, they have to be compiled. To do so, right click on the SmartSoft component project and choose "Compile SmartMDSO Project" (cf. figure 2.24).

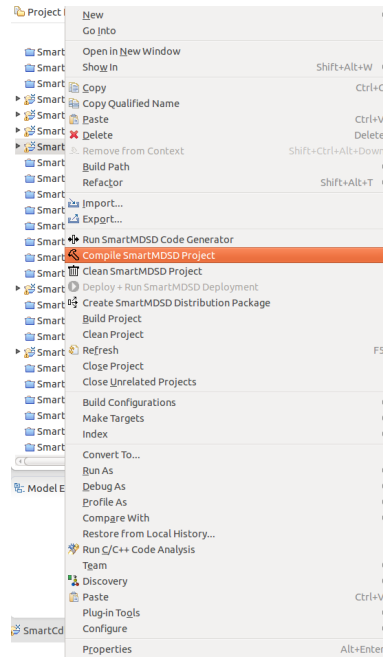


Figure 2.24. Compile SmartSoft Component Project

If the SmartSoft component project has to be compiled without the toolchain, navigate to the component and enter the following commands:

```
mkdir build
cd build
cmake ..
make
```

In every component project and every communication/coordination repository folder, you will find a cmake file CMakeLists.txt. This file can be adjusted to add component-specific library dependencies.

2.3.4.1. Add Additional Libraries

In order to add external library dependencies (which should provide a cmake package definition) the following lines have to be added to the file CMakeLists.txt:

```
FIND_PACKAGE(<library> REQUIRED <components>)
GET_PROPERTY(<library>_INCLUDE_DIRS DIRECTORY PROPERTY
INCLUDE_DIRECTORIES)
LIST(APPEND USER_INCLUDES ${<library>_INCLUDE_DIRS})
LIST(APPEND USER_LIBS ${<library>_LIBS})
```

If, for example, the libraries mrpt-base and mrpt-gui should be added, the following lines have to be used:


```
FIND_PACKAGE(MRPT REQUIRED base gui)
GET_PROPERTY(MRPT_INCLUDE_DIRS DIRECTORY PROPERTY
INCLUDE_DIRECTORIES)
LIST(APPEND USER_INCLUDES ${MRPT_INCLUDE_DIRS})
LIST(APPEND USER_LIBS ${MRPT_LIBS})
```

For further information on how to create cmake package-definitions for external libraries see: https://cmake.org/Wiki/CMake:How_To_Find_Libraries

System libraries (e.g. installed in /usr/lib) can be added as follows:

```
LIST(APPEND USER_LIBS "<library>")
```

The library libbluetooth, for example, can be added as follows:

```
LIST(APPEND USER_LIBS "bluetooth")
```

2.3.4.2. Add Compiler Flags

Additional compiler flags can be added to the CMakeLists.txt as follows:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -<compiler flag>")
```

For instance, in order to add the compiler flag "ENABLE_HASH" use this line:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DENABLE_HASH")
```

2.3.4.3. Add Your Own Source Files

All user source files inside of the src/ folder that have the ending .cc or .hh will be automatically included into the build process of the component. In order to use a custom subfolder inside of the src/ folder add the following lines to the CMakeLists.txt file:

```
FILE(GLOB SRCS src/<directory>/*.cc)
LIST(APPEND USER_SRCS ${SRCS})
LIST(APPEND USER_INCLUDES src/<directory>/)
```

An example can be found in the *SmartVisualization* component. The following lines were used to add the source files of the directory "visualization":

```
FILE(GLOB SRCS src/visualization/*.cc)
LIST(APPEND USER_SRCS ${SRCS})
LIST(APPEND USER_INCLUDES src/visualization/)
```

In case the subdirectory contains an own cmake project, use the following approach instead:

```
ADD_SUBDIRECTORY(<path>)
LIST(APPEND USER_INCLUDES <path>)
LIST(APPEND USER_LIBS <libraries>)
```

An example is shown in the component *SmartXsensIMUMTiServer*:

```
ADD_SUBDIRECTORY(${PROJECT_SOURCE_DIR}/src/xsensSDK)
LIST(APPEND USER_INCLUDES ${PROJECT_SOURCE_DIR}/src/xsensSDK/
Software_Development/CMTsrc)
LIST(APPEND USER_LIBS XSense)
```

Here is an example cmake project (for the XSense SDK):

```
SmartXsensIMUMTiServer/src/xsensSDK/CMakeLists.txt:

PROJECT(XSense)

FILE(GLOB SRCS ${PROJECT_SOURCE_DIR}/Software_Development/CMTsrc/
*.cpp)
FILE(GLOB HDRS ${PROJECT_SOURCE_DIR}/Software_Development/CMTsrc/
*.h)

INCLUDE_DIRECTORIES(${PROJECT_SOURCE_DIR}/Software_Development/
CMTsrc)

ADD_LIBRARY(${PROJECT_NAME} STATIC ${SRCS} ${HDRS})
```

2.3.5. Component Documentation

To add additional document to the component use the *.compdoc file. This file is located in the model-folder. The component can be generally described with the 'Description' keyword followed by a colon and a string:

```
Description : <text>
```

Furthermore information about licences, hardware requirements and the general purpose of the component can be added. To do so, the keywords 'Licence', 'HardwareRequirements' and 'Purpose' are used.

```
Licence : <text>
HardwareRequirements : <text>
Purpose : <text>
```

The different states and services of the component can also be described. A high level description of states can be added with the keyword 'State_neutral' or 'State_Mainstate'. Thereby the keyword 'State_neutral' is used to describe the neutral state and the keyword 'State_Mainstate' followed by the name of the state to describe a mainstate.

```
State_neutral : <text>
State_Mainstate <name> : <text>
```

Services are documented with the keyword 'Service' followed by the name of the service. The description and further information about their behavior in specific states are enclosed by curly braces:

```
Service <name> {
```

```

Description : <text>
State_neutral : <text>
State_Mainstate <name> : <text>
}

```

From the textual parameter documentation, the information provided in the *.compdoc file and the component model a documentation is generated with doxygen.

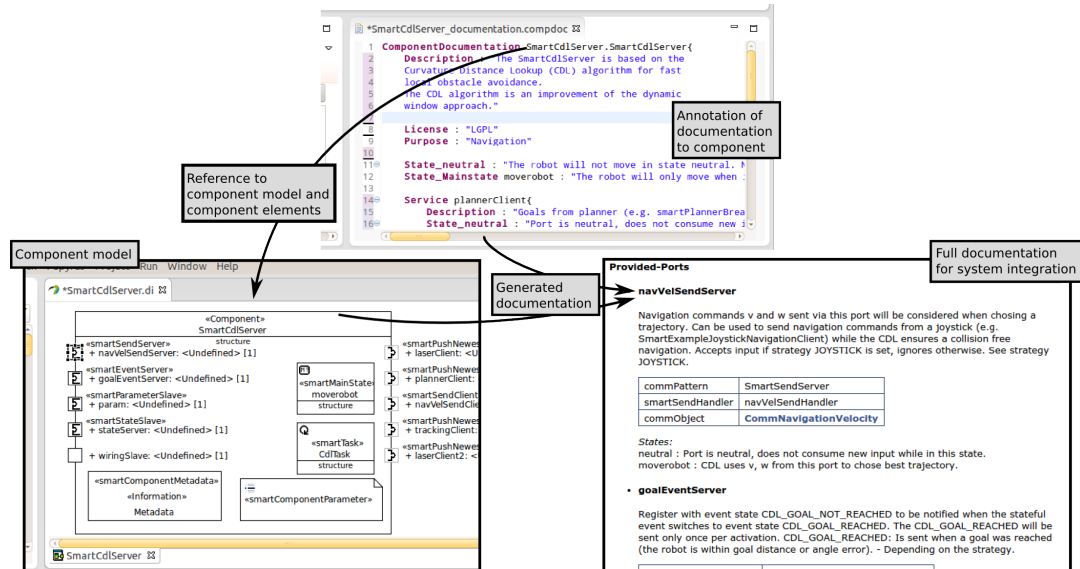


Figure 2.25. Information from the documentation and component model is transformed to a complete documentation (right) for later system integration which assists the system integrator during composition. [4]

2.4. System Composition View

In addition to the following description, the creation of a system configuration and a deployment model is shown in a screencast (cf. section 3.1.4 and section 3.1.6).

An application can be put together in the system composition View. For the application development, a new SmartSoft deployment project has to be created. To do so, choose "File->New->SmartSoft Deployment" (cf. figure 2.26), enter a name and choose a storage location. Typically the project is stored at "\$SMART_ROOT_ACE/src/deployments" and its name starts with "Deploy" followed by the application.

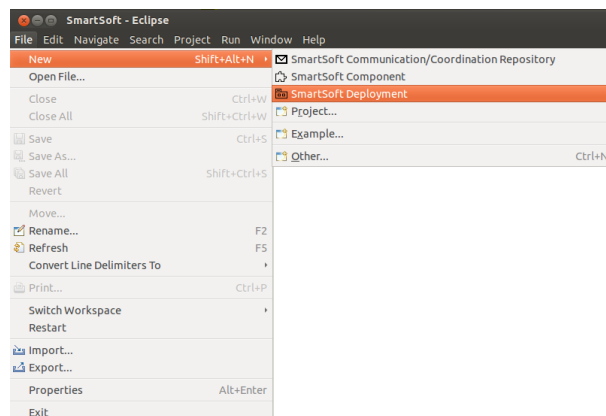


Figure 2.26. Create a new deployment project

Existing SmartSoft deployment projects can be imported by selecting "File->Import". In the appearing dialog choose "General->Existing Projects into Workspace".

The project contains a system configuration model and a deployment model. The system configuration model is used to connect and configure components. The deployment model is used to map the components of the system configuration model onto target hardware.

Before the system configuration or deployment model is altered, the graphical model of all used components should be closed. If the component model is changed while the SmartSoft deployment model is opened, the models might get out of sync.

2.4.1. System Composition Project

SmartSoft deployment projects contain various folders and files. For the development of applications the following are important:

- **model:** The model folder contains the graphical system configuration and deployment model (*.di).
- **src:** User-specific files for the deployment, typically hooks to execute commands before/after component start/stop. It is initially generated by the toolchain to provide a structure and standard files.
- **src-gen** Files generated by the SmartMDSO toolchain. Please do not modify these files, they will be overwritten each time the toolchain generator is run.

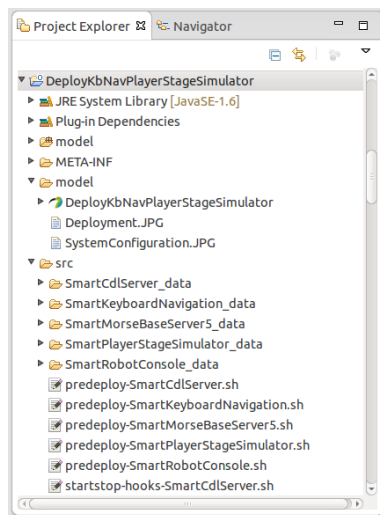


Figure 2.27. Structure of System Composition Projects

2.4.2. System Configuration

The system configuration is used to put together the application by (re)using building blocks such as components as well as behavior models. Thereby only the outer view on the hull (services) as well as the explicated configurations of the components are presented in the model [4].

The system configuration model is part of the system composition project and is located in the model-folder.

2.4.2.1. System Configuration Model

The system configuration is modeled graphically. It consists of a SmartSystemConfiguration which contains all building blocks. The name of the SmartSystemConfiguration must match the name of the System Composition Project. In order to model component instances inside the system configuration

the components have to be imported. To do so, right click on the system configuration model and choose "Import Components". The imported components are then listed in the model explorer and can be added to the model via drag and drop. The name of the component instances can be changed in the properties tab. After the component instance is added to the model no services are displayed. To show/hide the border items right click on the component instance and select "Filters" -> "Show/Hide Contents". Visible services can then be connected with a connector.

Figure 2.28 illustrates the modeling of the system configuration with the SmartMDS Toolchain.

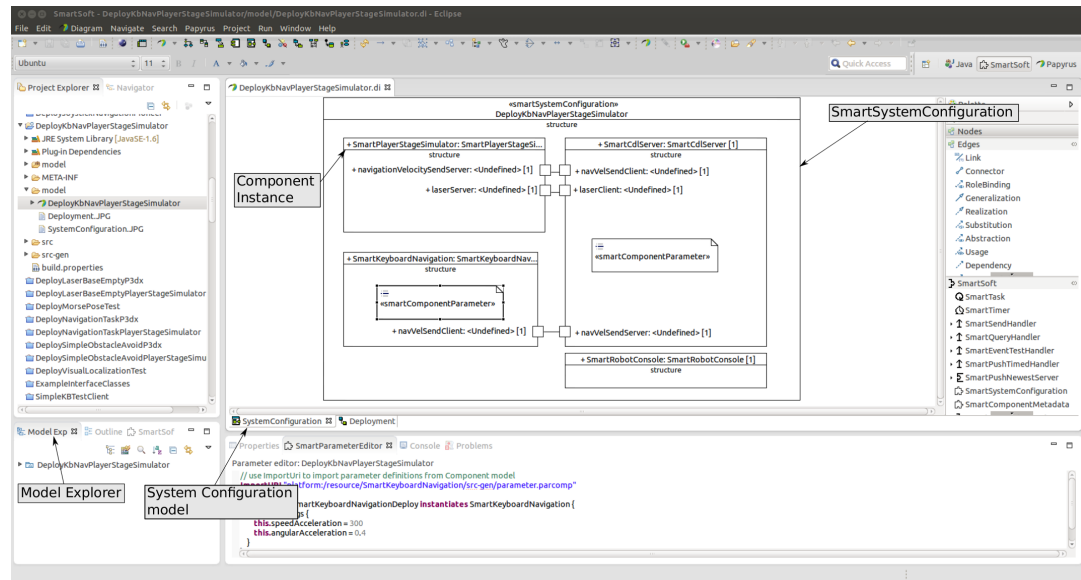


Figure 2.28. Modeling System Configuration

2.4.2.1.1. Change Connections

Existing connections between two component instances can be changed by selecting the connector element. The connection can then be altered through dragging the end of the connection to the new communication port.

2.4.2.1.2. Delete Components From the Model

If a component should be deleted from the model all instances must be removed (right click on the instance and choose "Delete Selected Element"). Then the imported component must be deleted in the model explorer (right click on the imported element and choose "Delete").

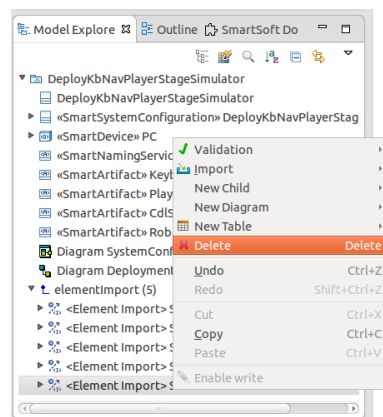


Figure 2.29. Delete Component from Model Explorer

Additionally the component has to be deleted from the Java Build Path. To do so, right click on the system composition project and select "Properties". On the left hand side tree of the upcoming dialog box select Java Build Path and switch to the "Projects" tab. The project can then be deleted by selecting the project and pressing the "Remove" button. Figure 2.30 shows the dialog box.

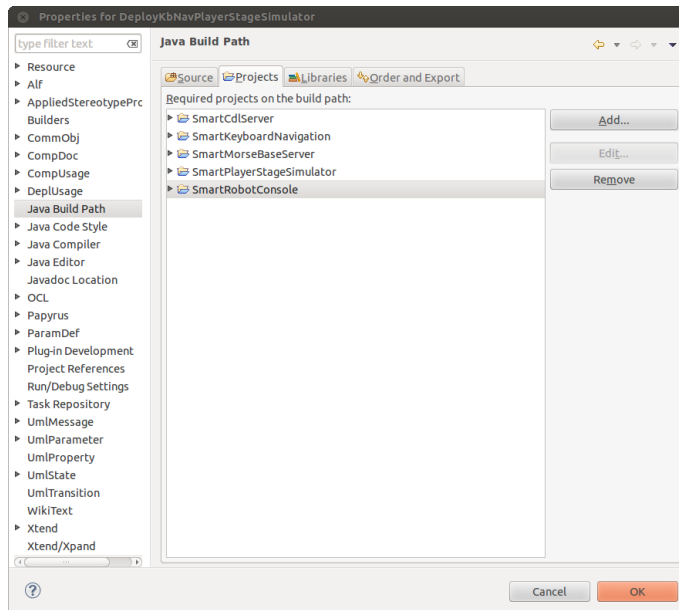


Figure 2.30. Delete Project from Java Build Path

2.4.2.2. Component Instance Configuration

To set the parameters of a component instance a `SmartComponentParameter` element has to be added to the model of the instance. The parameter is then set as follows:

```

ImportUri <path>

Deployment <instance name>Deploy instantiates <component name> {
  Param <name> {
    this.<attribute name> = <value>
  }
}

```

Parameters are set with the keyword 'Param' followed by the name of the parameter and the attributes which are enclosed by curly braces. The attributes which should be assigned with a value are selected with the `this.`-operator.

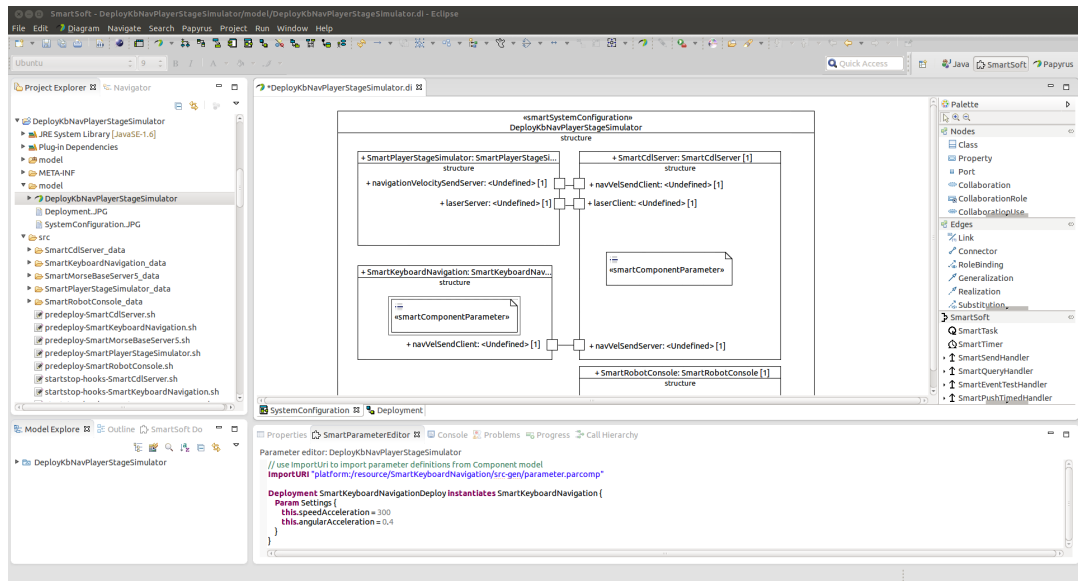


Figure 2.31. Configure component instance

2.4.3. System Deployment

2.4.3.1. Deployment model

The system deployment model is part of a SmartSoft system composition project and is located in the model-folder. It is modeled graphically and contains SmartDevices, SmartArtifacts and a SmartNamingService.

An element is added to the deployment model by selecting the element in the tool palette and a left mouse click on the model. Due to a bug in PapyrusUML the stereotypes are not applied correctly. Therefore, the stereotypes have to be added manually. To this select the element and add the stereotype by clicking the "+"-button in the Properties tab (Properties->Profile). Figure 2.33 illustrates the modeling of the system deployment with the SmartMDS Toolchain.

- *SmartDevice*: The SmartDevice element is used to model target computers to which the components are deployed. The following settings can be made:
 - *ip*: Specifies the ip-address of the device. Default is 127.0.0.1
 - *loginName*: Specifies the login name of the device. If the value is an empty string, the login name will be the name of the current user logged in.
 - *deploymentDirectory*: Specifies the storage location of the deployment. Default is ~/tmp/
- *SmartArtifacts*: SmartArtifacts are used to model component instances which are already modeled in the system configuration model. For every component instance in the system configuration model a SmartArtifact has to be added. The modeled SmartArtifacts are distributed to a target computer with a deployment arrow from the target computer to the component instance. The following settings can be made:
 - *utilizedComponentInstance*: Specifies the component instance. To select the component instance press the "..."-Button on the right, choose the instance in the appearing dialog and press the OK button.
- *SmartNamingService*: The deployment model must contain exactly one SmartNamingService which is distributed to a target computer with a deployment arrow. The following settings can be made:

- *port*: Specifies the port of the naming service. The default value is suitable for most use-cases.

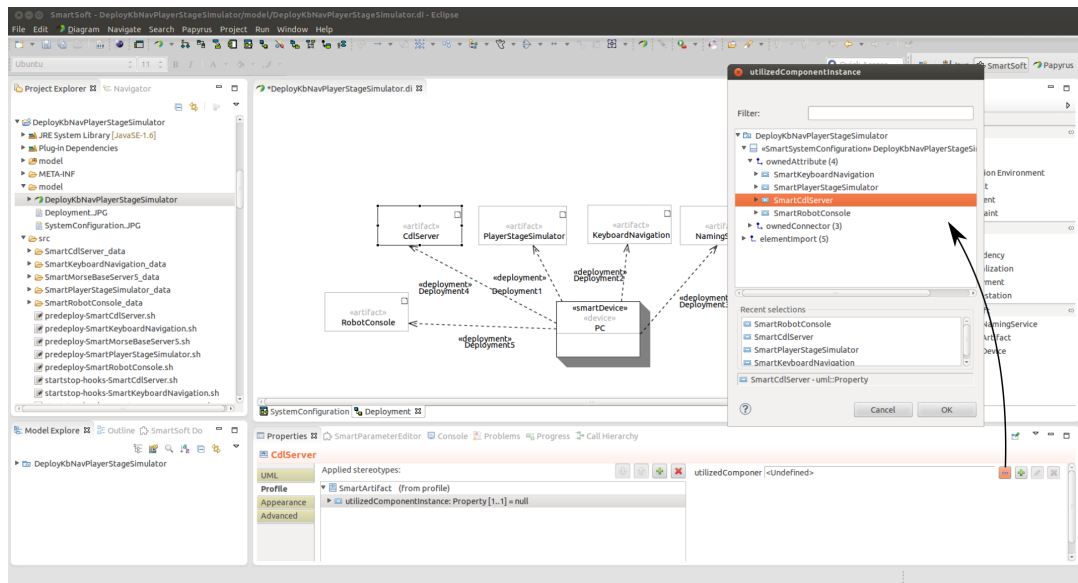


Figure 2.32. Selecting a utilized component instance.

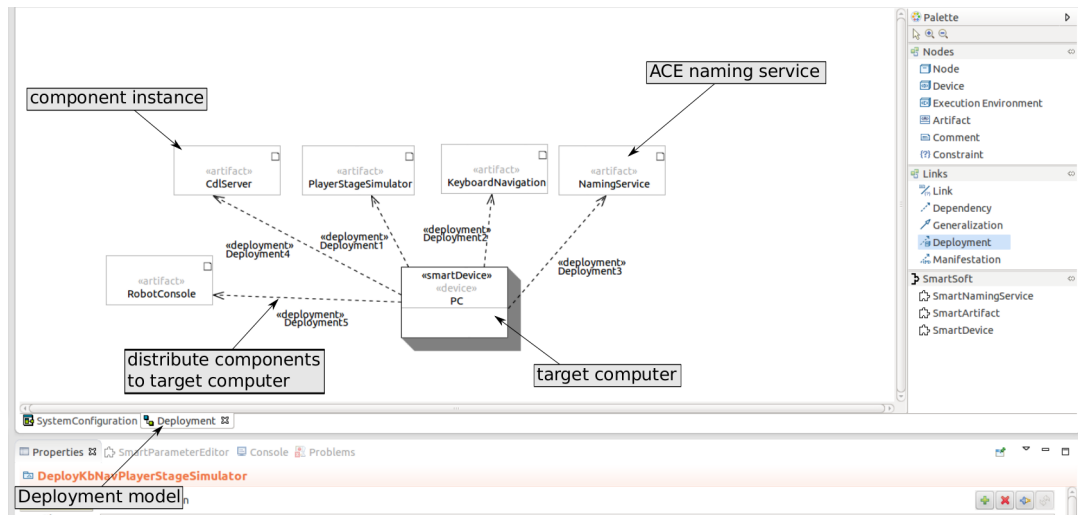


Figure 2.33. Modeling System Deployment

2.4.3.2. Code Generation

After the system configuration model and the deployment model are finished the code generator can be started. To do so, right click on the SmartSoft deployment project and choose "Run SmartMDS Code Generator" (cf. figure 2.34).

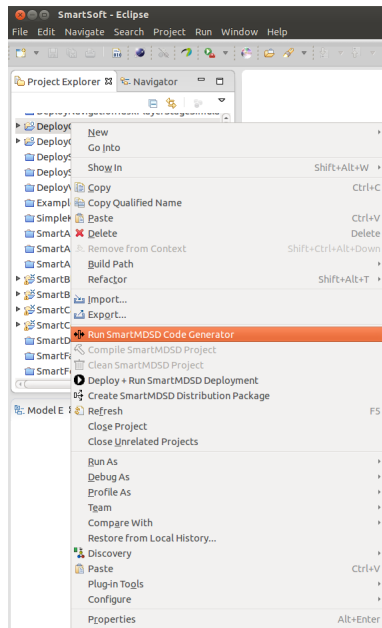


Figure 2.34. SmartSoft Deployment Code Generator

2.4.3.3. Target Considerations

The SmartMDSO Toolchain does not support cross-compilation. Therefore, the system deployment has to be developed and executed on the same architecture.

All SmartSoft dependencies are deployed automatically. If additional libraries should be used, they have to be added manually. To do so, adjust the script `predeploy.sh`. This script is generated for each component in the model and is run prior to the deployment of the component. The script can be used to add SmartSoft libraries that shall be deployed to the target device. These libraries will be searched in `$SMART_ROOT/lib`. The libraries are added as follows:

```
DEPLOY_LIBRARIES="$DEPLOY_LIBRARIES <library>.so"
```

2.4.3.4. Deploying Additional Files

If components should be deployed along with additional data files they have to be added after the code generation. After the code generation there exists a data-folder for every component instance inside the `src`-folder to which additional data files can be added. The screencast in section 3.1.6 demonstrates the deployment of components along with additional data files.

2.4.3.5. Start-Stop-Hooks

For every component in the model, a `startstop-hooks` script is generated. This script provides methods to call custom commands pre/post of starting/stopping the component during launch on the device. This script is being executed on the target device where the component is running. For example the script can be used to start and stop the morse simulator automatically.

2.4.3.6. Predeploy Infrastructure

For every component in the model, a `predeploy` script is generated. This script is run prior to deployment of component. E.g. use this script to collect data-files and copy them to `src/<COMPONENT>_data`, etc.

2.4.3.7. Deploying the Application

Precondition:

Deployment is done via scp which needs a password. We suggest to setup an ssh-key: before deploying an application create a ssh-key for the target device deployment once. To do so, enter in a terminal:

```
ssh-keygen
```

Then, use ssh-copy-id to transfer the key to the remote machine:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub user@host
```

You should make sure that you can now login to the host without a password.

Deploying the application:

To deploy the application from the SmartMDS Toolchain right click on the SmartSoft deployment project and choose "Deploy + Run SmartMDS Deployment". After the deployment has finished, the application can be started (cf. section 2.4.4.1). If the toolchain was started with a terminal, the ssh yes/no as well as the password input are displayed in the terminal. If the toolchain was started with the icon, these inputs are displayed in a dialog.

To deploy the application without the toolchain, open a terminal and navigate to the deployment folder. Then enter the following command:

```
bash src-gen/deploy-all.sh
```

2.4.4. Running the Application

2.4.4.1. Running the Application from Toolchain

The deployment and execution of an application is also demonstrated in a screencast (section 3.1.5).

To run the application from the toolchain right click on the SmartSoft deployment project and choose "Deploy + Run SmartMDS Deployment" (cf. figure 2.35).

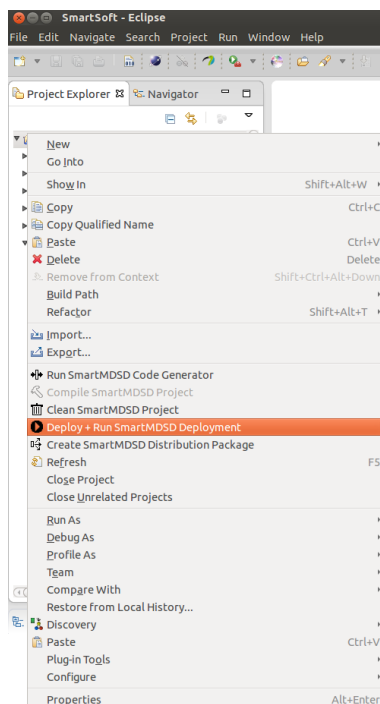


Figure 2.35. Running the application from the toolchain

As soon as the deployment finishes press the "Yes"-Button on the appearing dialog. A terminal (Global Scenario Control) is opened automatically (cf. figure 2.36). To start the application choose "menu-start".

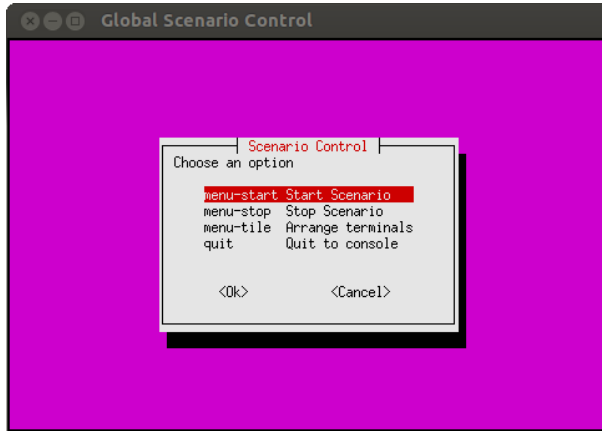


Figure 2.36. Global Scenario Control

To stop the application choose "menu-stop" in the global scenario control. After all components have closed choose "quit".

2.4.4.2. Running the Application without the Toolchain

You will find the deployment at the target folder that you specified in the deployment project. Default is `~/tmp/<DEPLOYMENT-PROJECT-NAME>.deploy`. To start or stop the application enter:

```
bash start-PC.sh start
```

or

```
bash start-PC.sh stop
```

2.4.4.3. Component Output and Log Files

The components will be started in an terminal window. In case a component closes unexpectedly, the window stays open and you can inspect the component output. Additionally, the output of all components is kept in log-files which will be compressed once you close the deployment (`*-logs-*.tar.gz`).

2.5. Tips and Tricks

2.5.1. SmartSoft Full Build of Source Tree

It is possible to compile the SmartSoft Kernel and all SmartSoft projects that are located in the directory of the smartsoft repository checkout (`$SMART_ROOT_ACE`). To do so, open a terminal and enter the following commands:

```
cd $SMART_ROOT_ACE
mkdir build
cd build
cmake ..
make
```

To extend the list of components build in this run one can change CMakeLists.txt file in \$SMART_ROOT_ACE/src/components/.

2.5.2. SmartSoft and the RaspberryPi

There is (experimental) support of the ARM architecture and the RaspberryPi by the SmartSoft infrastructure and components

However, the SmartMDS Toolchain currently does not support cross-compilation. The SmartMDS Toolchain is in most cases used on a standard PC/x86 platform, while the RaspberryPi is based on an ARM architecture. It is therefore not possible to use the SmartMDS Toolchain with the RaspberryPi. This section gives instructions on how to use the components and applications created with the SmartMDS Toolchain manually on the RaspberryPi.

2.5.2.1. Preconditions

- SmartSoft must be installed on the RaspberryPi. Use the provided installation script that comes with SmartSoft; it automatically detects the RaspberryPi and adjusts the installation routine. Experimental support is available for Raspbian (confirmed with Raspbian Jessie / 2015-11-21).
- All described instructions must be executed from within the RaspberryPi graphical user interface / X-Session or from ssh remote login with x forwarding option "ssh -X".
- Optional: We recommend to make sure that you can login from RaspberryPi to localhost without a password. We recommend to use ssh-copy-id to the PI localhost.
- Deployments will be made from the RaspberryPi to the RaspberryPi. Note that you can only deploy to RaspberryPi devices. A deployment with two devices of different architectures (e.g. one RaspberryPi and one x86) is not supported.
- Note that the user and IP of the device in the deployment project have to align with the viewpoint of the RaspberryPi. For most cases with one device, you can leave the IP at its default 127.0.0.1

2.5.2.2. Step by Step Instructions

- Create / implement your projects as usual in the toolchain. Run the code generator for the deployment project in SmartMDS Toolchain.
- Make sure that all projects (deployment, components, comm-objs) are available on the RaspberryPi. For example, use versioning control like SVN/GIT to keep them in sync, as you might want to continue development on your development host (PC).
- Compile all communication object projects and component projects manually. (Eventually delete build/ from component folder, if you copied them from your development host)

```
$ cd <comm-obj or component directory>
$ mkdir build
$ cmake ..
$ make
```

- Edit DEPLOYMENT-PROJECT/src-gen/referenced-projects and correct the absolute paths of the component directories. This file is generated by the SmartMDS Toolchain but contains paths from the development computer. Depending on your setup, these paths may be wrong and you need to correct them to match your RaspberryPi setup.
- Now trigger the deployment action by executing the deployment script from the deployment project directory.:

```
$ bash src-gen/deploy-all.sh
```

- You will find the deployment at the target folder that you specified in the deployment project. Default is `~/tmp/<DEPLOYMENT-PROJECT-NAME>.deploy`
- To start and stop the deployment, proceed as described in 2.4.4.2.

2.5.3. Delete Model Elements

If a model element should be deleted, you should not use the delete key of the keyboard. The delete key has two different assignments. It can delete or hide the selected element. Due to this double assignment the model element is not deleted reliably. To delete a model element, right click on the element and choose "Delete Selected Element". To make sure that elements are not part of the model anymore, you can check so in the model explorer.

2.5.4. Common Error Messages

The SmartMDS Toolchain contains basic checks of your models. The following sections list common error messages that might come up in such cases during code generation or compilation. They are presented along with suggestions how to solve them. The error messages are shown in the Console-tab of the toolchain (cf. figure 2.37).

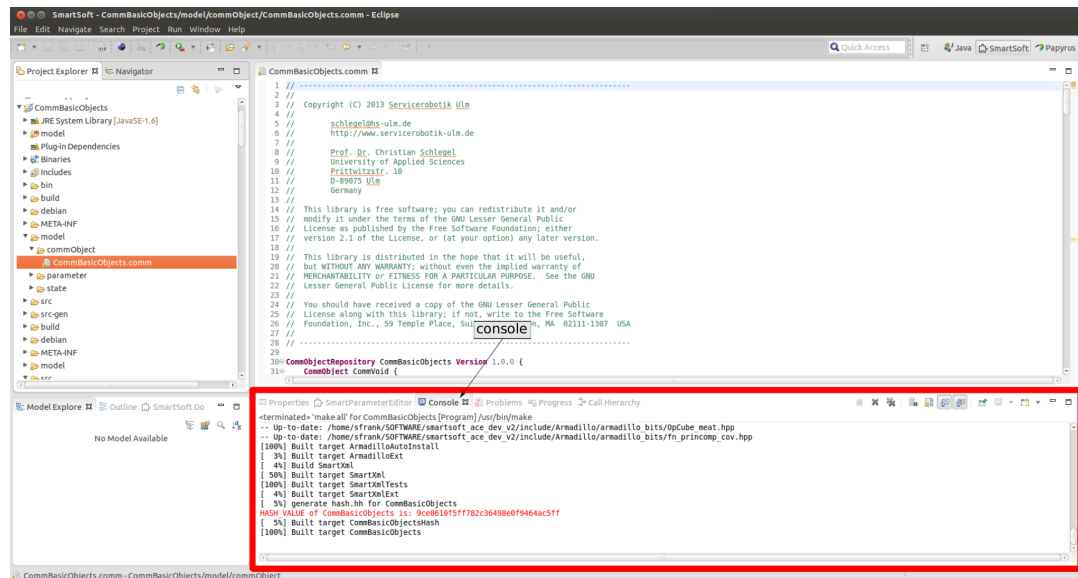


Figure 2.37. Console tab of the SmartMDS Toolchain

2.5.4.1. Component Development View

Error during Code generation:

- **[ERROR]: No Metadata element defined. Please add one.: <component name>(Element: ERROR:No Metadata element defined. Please add one.: <component name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no SmartComponentMetadata element was added to the component. To solve the problem add a SmartComponentMetadata element to the component model.

- **[ERROR]: 2 Metadata elements defined. Only one allowed.: <component name>(Element: ERROR:2 Metadata elements defined. Only one allowed.: <component name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if more than one SmartComponentMetadata element was added to the component. To solve the problem remove the superfluous SmartComponentMetadata elements from the component model.

- **[ERROR]: No valid communication object assigned for this service: <service name>(Element: ERROR:No valid communication object assigned for this service: <service name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no communication object is assigned to the service <service name>. To solve the problem assign a communication object to the corresponding service.

- **[ERROR]: No valid smart<pattern>Handler set: <service name>(Element: ERROR:No valid smart<pattern>Handler set: <service name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no valid handler is assigned to the service. To solve the problem assign the corresponding handler to the service.

- **[ERROR]: No SmartComponentParameter defined. Please add one.: <component name>(Element: ERROR:No SmartComponentParameter defined. Please add one.: <component name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if a smartParameterSlave but no SmartComponentParameter element was added to the model. To solve the problem add a SmartComponentParameter to the model.

- **[ERROR]: 2 Smart<pattern>Slaves defined. Only one allowed.: <component name>(Element: ERROR:2 Smart<pattern>Slaves defined. Only one allowed.: <component name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if more than one smartWiringSlave, smartParameterSlave or smartStateSlave was added to the model. To solve the error, delete the superfluous communication ports.

Error during compilation:

- **<path to component>/src-gen/<component name>.hh:25:34: fatal error: <communication object name>.hh: No such file or directory**

This error message is shown if the communication object was not compiled. To solve the problem compile the corresponding SmartSoft communication/coordination Repository (cf. section 2.2.4).

- **<path to component>/src-gen/<component name>.hh:31:32: fatal error: <handler name>.hh: No such file or directory**

This error message is shown if a handler but no corresponding communication port was added to the model. To resolve the problem add the corresponding communication port.

2.5.4.2. System Composition View

Error during code generation:

- **[ERROR]: No SmartNamingService defined: <deployment name>(Element: ERROR:No SmartNamingService defined: <deployment name> (line : null); Reported by: -UNKNOWN-)**

This error is shown if no SmartNamingService is modeled in the deployment model. To solve the problem add a SmartNamingService element and connect it to the device.

- **[ERROR]: No stereotype assigned. Please assign SmartArtifact or SmartNamingService: <artifact name>(Element: ERROR:No stereotype assigned. Please assign SmartArtifact or SmartNamingService: <artifact name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no stereotype is assigned to a SmartArtifact element in the deployment model. To solve the problem assign a stereotype to the SmartArtifact element by pressing the "+"-button in the properties tab.

- **[ERROR]: Artifact does not reference component (property: utilizedComponentInstance): <artifact name>(Element: ERROR:Artifact does not reference component (property: utilizedComponentInstance): <artifact name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no component instance was assigned to a SmartArtifact. To solve the problem assign a component instance to the artifact in the properties-tab.

- **[ERROR]: No stereotype assigned. Please assign SmartDevice: <device name>(Element: ERROR:No stereotype assigned. Please assign SmartDevice: <device name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if no stereotype was assigned to a SmartDevice element. To solve the problem assign the stereotype "SmartDevice" to the SmartDevice element by pressing the "+"-Button in the properties tab.

- **[ERROR]: Only the stereotype SmartDevice is allowed.: <device name>(Element: ERROR:Only the stereotype SmartDevice is allowed.: <device name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if a stereotype other than "SmartDevice" or no stereotype was assigned to a SmartDevice element. To solve the problem assign the stereotype "SmartDevice" to the SmartDevice element.

- **[ERROR]: No loginName set: <device name>(Element: ERROR:No loginName set: <device name> (line : null); Reported by: -UNKNOWN-)**

This error message is shown if the loginName of a SmartDevice element is NULL. To solve the problem change the value of the login name (an empty string is allowed).

Error during the deployment:

- **ERROR: FILES ARE MISSING FROM THE DEPLOYMENT (see above). Did you compile all components?**

This error message is shown if at least one component, which is used in the deployment, was not compiled. To solve the problem make sure all used components are compiled (cf. section 2.3.4).

Chapter 3. Tutorials

This chapter describes step-by-step tutorials (in written form and as video screencasts) to guide the reader through all views of the SmartMDS Toolchain to develop a practical example from scratch.

3.1. Video Tutorials

This series of screencasts demonstrates the use of the SmartMDS Toolchain. It can be used as a walk-through tutorial through all stages of the development process and major functionalities of the toolchain. Unless otherwise mentioned, the sequence of videos is in chronological order and each extends or continues its predecessor.

3.1.1. Tutorial 1: Modeling of Communication Objects

This video demonstrates the modeling of Communication Objects using the SmartMDS Toolchain.

The application in mind behind all examples is a robot that drives around while avoiding any obstacles. In this video, a Communication Object Repository (CommBasicObjects) is created with a simple Communication Object CommNavigationVelocity that can be used to command navigation instructions to a robot. The video also demonstrates how to model nested and more complex Communication Objects (CommMobileLaserScan).

This video shows only an excerpt of the modeling of the repository CommBasicObjects which is readily available within the SmartSoft distribution.

Link to the Video: [https://www.youtube.com/watch?v=0x_nhFatO5w].

3.1.2. Tutorial 2: Definition of a Parameter Set

This video demonstrates the modeling of a parameter using the SmartMDS Toolchain.

The application in mind behind all examples is a robot that drives around while avoiding any obstacles. The parameter represents a configurable maximum velocity of a robot. This parameter can later be instantiated by components. The maximum speed can then be configured through the parameter service.

(as shown in: [5] see *Slides* [http://servicerobotik-ulm.de/drupal/sites/default/files/2014_2014-07-13-RSS2014-Tutorial-Website.pdf])

Link to the Video: [<https://www.youtube.com/watch?v=2U4KxSgwtqY>].

3.1.3. Tutorial 3: Component Development

This video demonstrates the modeling and implementation of a component using the SmartMDS Toolchain.

The application in mind behind all examples is a robot that drives around while avoiding any obstacles. The component receives laser scans. A simple obstacle avoidance algorithm outputs values for speed and direction. The component then thresholds the maximum speed according to a variation point (parameter "v_x", modeled in a previous video) before providing the navigation commands through one of its services.

This parameter "v_x" can be configured during runtime of the component through its parameter service.

(as shown in: [5] see *Slides* [http://servicerobotik-ulm.de/drupal/sites/default/files/2014_2014-07-13-RSS2014-Tutorial-Website.pdf])

Link to the Video: [<https://www.youtube.com/watch?v=chyRCu4FCbs>].

3.1.4. Tutorial 4: System Configuration and Deployment Model

This video demonstrates the creation of system configuration and deployment model using the SmartMDSD Toolchain.

The application in mind behind all examples is a robot that drives around while avoiding any obstacles. The scenario: a robot shall drive and avoid obstacles. It reuses (existing) components SmartLaserObstacleAvoid (see previous screencast), SmartLaserLMS200Server (laser ranger) and SmartPioneerBaseServer (robot).

The system configuration model models the connection and configuration of components. The deployment model models the distribution of components on hardware.

(as shown in: [5] see *Slides* [http://servicerobotik-ulm.de/drupal/sites/default/files/2014_2014-07-13-RSS2014-Tutorial-Website.pdf])

Link to the Video: [<https://www.youtube.com/watch?v=y-S33qaeNfI>].

3.1.5. Tutorial 5: Deploying and Running an Application

This video demonstrates the deployment and execution of an application developed using the SmartMDSD Toolchain.

The application in mind behind all examples is a robot that drives around while avoiding any obstacles. The application (laser obstacle avoidance from a previous video) is deployed using SSH. A remote session on the robot is established in order to run it. The robot will first drive with a maximum velocity of 600m/s (as configured in system configuration). Later, run-time configuration is used to change the maximum velocity of the component to 200 and back to 600 every 5s via the parameter service and explicated variation point `v_x`.

(as shown in: [5] see *Slides* [http://servicerobotik-ulm.de/drupal/sites/default/files/2014_2014-07-13-RSS2014-Tutorial-Website.pdf])

Link to the Video: [https://www.youtube.com/watch?v=OZcC4ipt_BM].

3.1.6. Tutorial 6: Deployment of components along with additional files

This video demonstrates the deployment of components along with additional data files. It bases on a existing deployment diagram (*DeployNavigationTaskPlayerStageSimulator* [<http://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/deployments/DeployNavigationTaskPlayerStageSimulator/>]) and covers the steps from tutorial 5 with deploying additional files.

The example shows basic navigation components performing obstacle avoidance using the SmartCdIserver component. The CDL algorithm makes use of local lookup files which need to be deployed to the target host as well. This video shows how to deploy these files from within a deployment project of the SmartMDSD toolchain.

This video also makes use of the Player Stage simulator component.

Link to the Video: [<https://www.youtube.com/watch?v=JTww2aSBxac>].

3.2. Step by Step: Robot navigation

3.2.1. Introduction

This step by step tutorial shows a complete walkthrough through all steps of the SmartMDS Tool-chain to develop a robot navigation. The robot will be simulated in a 2D environment and shall be controlled with a keyboard. Section 3.2.2 shows the modeling and implementation of the *SmartKeyboardNavigation* component. After the component is developed two different examples of System Compositions are made in section 3.2.3. The first allows the user to control the robot with the keyboard. The second adds obstacle avoidance to the first example. All required components are available online.

The example consists of the following components:

- *SmartKeyboardNavigation* [<http://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/components/SmartKeyboardNavigation/>], which will be developed in this tutorial. It is also available online.
- *SmartPlayerStageSimulator* [<http://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/components/SmartPlayerStageSimulator/>]
- *SmartCdlServer* [<http://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/components/SmartCdlServer/>]
- *SmartRobotConsole* [<http://sourceforge.net/p/smartsoft-ace/code/HEAD/tree/trunk/src/components/SmartRobotConsole/>]

The *SmartKeyboardNavigation* component is used to control the robot with the keyboard. The *SmartPlayerStageSimulator* component simulates the robot in a 2D environment. The *SmartCdlServer* component is used for obstacle avoidance and the *SmartRobotConsole* is used to set parameters during runtime.

3.2.2. Component Development (SmartKeyboardNavigation)

In this section the *SmartKeyboardNavigation* component will be modeled and implemented. To do so, a new SmartSoft Component has to be created. The name of this new component should be *SmartKeyboardNavigation*.

After creating the project the component itself has to be modeled. The component should be able to catch the keyboard input and send the current velocity of the robot. The keyboard input will be received and processed with a few lines of C++ code. This code will run in an user thread, therefore the SmartTask "KeyboardInputTask" has to be added to the component. The thread should be executed approx. every 500ms, therefore the KeyboardInputTask should be modeled periodically with a period of 500ms (see figure 3.2).

To control the velocity of the robot a SmartSendClient "navVelSendClient" is modelled. The velocity is sent with the "CommNavigationVelocity" Communication Object. This Communication Object can be found in the CommBasicObjects repository for importing. The serverName and serviceName the new SmartSendClient has to connect to can be left blank. They will be set in the System Configuration in the next step. Finally the SmartComponentMetadata has to be added to the component.

In summary the *SmartKeyboardNavigation* component consists of:

- a SmartTask (name: KeyboardInputTask, isPeriodic: true, timeUnit: ms, period: 500)
- a SmartSendClient (name: navVelSendClient, commObject: CommNavigationVelocity)
- a SmartComponentMetadata

Figure 3.1 shows the modeled component.

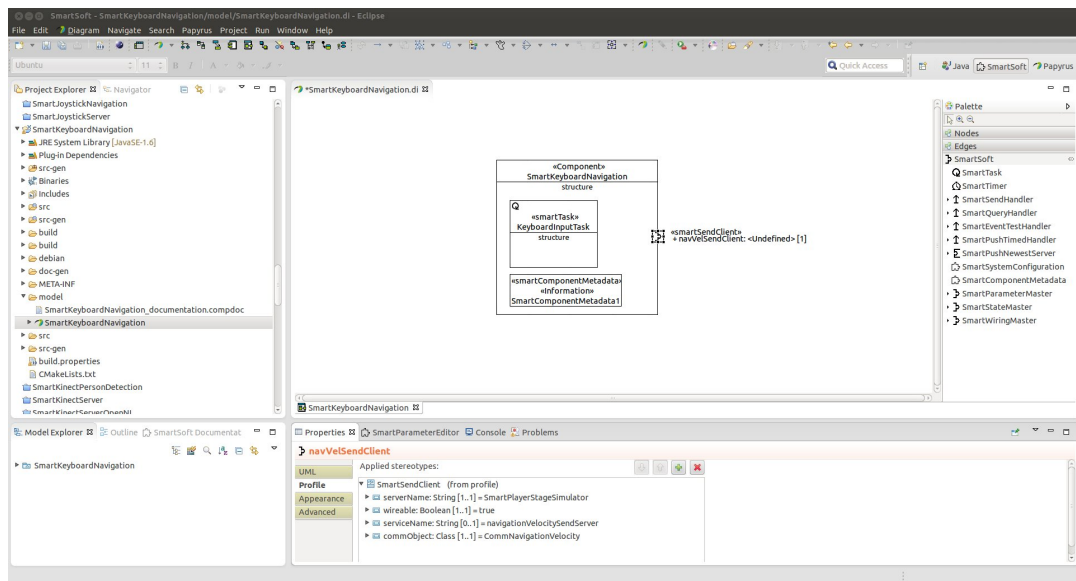


Figure 3.1. SmartKeyboardNavigation

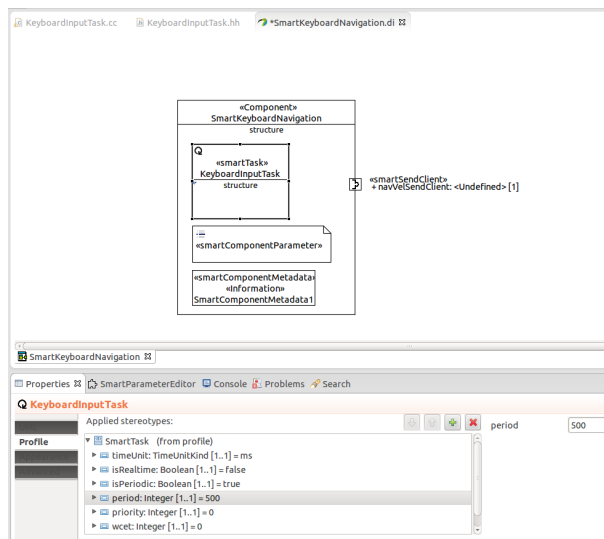


Figure 3.2. KeyboardInputTask task with its timing parameters.

After the component is modeled the c++ code of the component hull with its inner and outer interfaces has to be generated to start the implementation of the keyboard navigation component. To do so, the user code has to be written in the generated "KeyboardInputTask.cc" and "KeyboardInputTask.hh" user code files.

First of all it has to be possible to detect a keyboard input. To do so, the function *kbhit()*¹ is implemented in the file "KeyboardInputTask.cc":

```
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
```

¹Implementation on kbhit taken from: <http://cboard.cprogramming.com/c-programming/63166-kbhit-linux.html>

```
#include <fcntl.h>

int kbhit(void)
{
    struct termios oldt, newt;
    int ch;
    int oldf;

    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

    ch = getchar();

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);

    if(ch != EOF)
    {
        ungetc(ch, stdin);
        return 1;
    }

    return 0;
}
```

If a key is pressed the velocity of the robot has to be adjusted. The current speed control values are stored in task member variables "x" and "omega". These variables are defined in the "KeyboardInputTask.hh" file:

```
class KeyboardInputTask : public KeyboardInputTaskCore
{
public:
    KeyboardInputTask(CHS::SmartComponent *comp);
    virtual ~KeyboardInputTask();

    virtual int on_entry();
    virtual int on_execute();
    virtual int on_exit();

    double x;
    double omega;
};
```

Depending on the pressed key the speed or the heading velocity has to be increased or decreased. To do so, the following code has to be executed each time the task is executed.

```
int KeyboardInputTask::on_execute()
{
```

```
if(kbhit()) {
    char c = getchar();

    /*Arrow keys*/
    if (c == '\033') { // if the first value is esc
        getchar(); // skip the [
        switch(getchar()) { // the real value
            case 'A':
                std::cout << "Accelerating" << std::endl;
                x += 150;
                break;
            case 'B':
                std::cout << "Decreasing speed" << std::endl;
                x -= 150;
                break;
            case 'C':
                std::cout << "Shifting steering to right" << std::endl;
                omega -= 0.2;
                break;
            case 'D':
                std::cout << "Shifting steering to left" << std::endl;
                omega += 0.2;
                break;
        }
    }
    /*WASD*/
} else if(c == 'w') {
    std::cout << "Accelerating" << std::endl;
    x += 150;
} else if (c == 'd') {
    std::cout << "Shifting steering to right" << std::endl;
    omega -= 0.2;
} else if (c == 'a') {
    std::cout << "Shifting steering to left" << std::endl;
    omega += 0.2;
} else if (c == 's') {
    std::cout << "Decreasing speed" << std::endl;
    x -= 150;
} else if (c == 'q') {
    std::cout << "Emergency fullstop" << std::endl;
    x = 0;
    omega = 0;
}
}

CommBasicObjects::CommNavigationVelocity vel;
vel.set_vX(x);
vel.set_omega(omega);

CHS::StatusCode status = COMP->navVelSendClient->send(vel);
}
```

If a key is pressed the x or omega variable will be adjusted. The velocity values are then stored in the CommNavigationObject "vel" and send via the SmartSendClient "navVelSendClient".

In the example above the speed of the robot is increased or decreased by 150[mm/s] and the heading velocity is increased or decreased by 0.2[rad/s]. To make those values configurable store them in

SmartComponentParameters. For this purpose a SmartComponentParameter has to be added to the component model. Figure 3.3 shows the *SmartKeyboardNavigation* component with an added SmartComponentParameter.

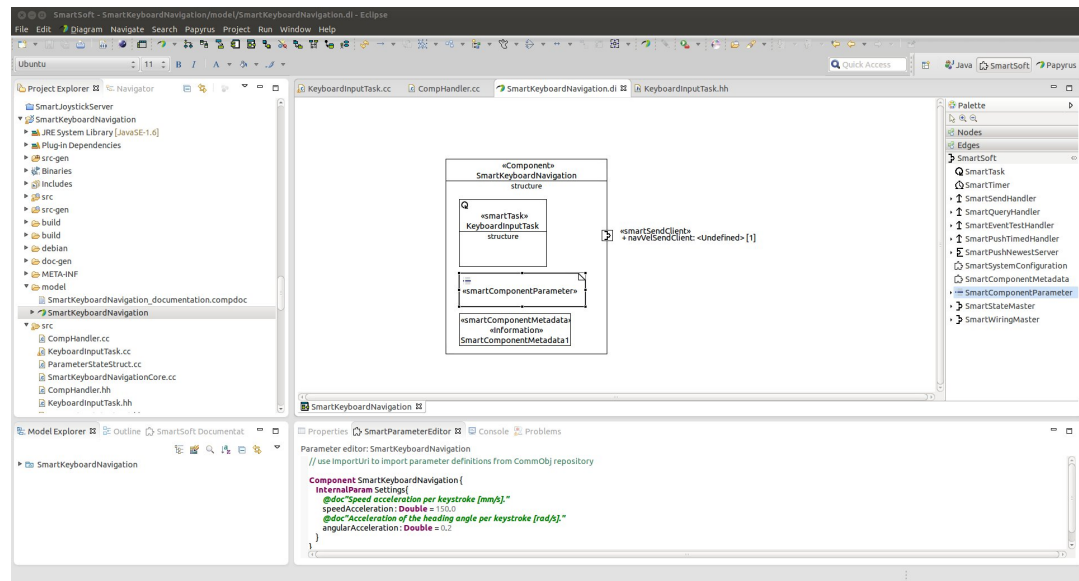


Figure 3.3. Adding parameters

The internal parameters speedAcceleration and angularAcceleration are added:

```
InternalParam Settings{
    speedAcceleration : Double = 150.0
    angularAcceleration : Double = 0.2
}
```

After adding the parameters the code has to be regenerated. Additionally, the code of the "KeyboardInputTask.cc" has to be adapted. The values 150 and 0.2 have to be exchanged by the corresponding parameters:

```
int KeyboardInputTask::on_execute()
{
    ParameterStateStruct::SettingsType localstate = COMP-
>getGlobalState().getSettings();
    if(kbhit()) {
        char c = getchar();

        /*Arrow keys*/
        if (c == '\033') { // if the first value is esc
            getchar(); // skip the [
            switch(getchar()) { // the real value
                case 'A':
                    std::cout << "Accelerating" << std::endl;
                    x += localstate.getSpeedAcceleration();
                    break;
```

```
    case 'B':
        std::cout << "Decreasing speed" << std::endl;
        x -= localstate.getSpeedAcceleration();
        break;
    case 'C':
        std::cout << "Shifting steering to right" << std::endl;
        omega -= localstate.getAngularAcceleration();
        break;
    case 'D':
        std::cout << "Shifting steering to left" << std::endl;
        omega += localstate.getAngularAcceleration();
        break;
    }
    /*WASD*/
} else if(c == 'w') {
    std::cout << "Accelerating" << std::endl;
    x += localstate.getSpeedAcceleration();
} else if (c == 'd') {
    std::cout << "Shifting steering to right" << std::endl;
    omega -= localstate.getAngularAcceleration();
} else if (c == 'a') {
    std::cout << "Shifting steering to left" << std::endl;
    omega += localstate.getAngularAcceleration();
} else if (c == 's') {
    std::cout << "Decreasing speed" << std::endl;
    x -= localstate.getSpeedAcceleration();
} else if (c == 'q') {
    std::cout << "Emergency fullstop" << std::endl;
    x = 0;
    omega = 0;
}
}
vel.set_vX(x);
vel.set_omega(omega);

CHS::StatusCode status = COMP->navVelSendClient->send(vel);
}
```

With this the component modeling and implementation is done and the new component can be build (cmake).

3.2.3. System Composition

3.2.3.1. System Composition 1: Simple Scenario

After all components are modeled and compiled they can be used in a system configuration and deployment model. In this scenario a robot should be controlled with a keyboard. To do so, a new SmartSoft Deployment Project has to be created and the following components have to be imported into the System Configuration:

- SmartKeyboardNavigation
- SmartPlayerStageSimulator

Instances of these components have to be created in the System Configuration. Additionally, the services

- "navigationVelocitySendServer" of the *SmartPlayerStageSimulator* and
- "navVelSendClient" of the *SmartKeyboardNavigation*

have to be enabled and connected. To be able to change the values of the parameters a *SmartComponentParameter* is added to the instance of the *SmartKeyboardNavigation* component. In this example the values are doubled.

```
Param Settings {
  this.speedAcceleration = 300
  this.angularAcceleration = 0.4
}
```

Figure 3.4 illustrates the modeled System Configuration.

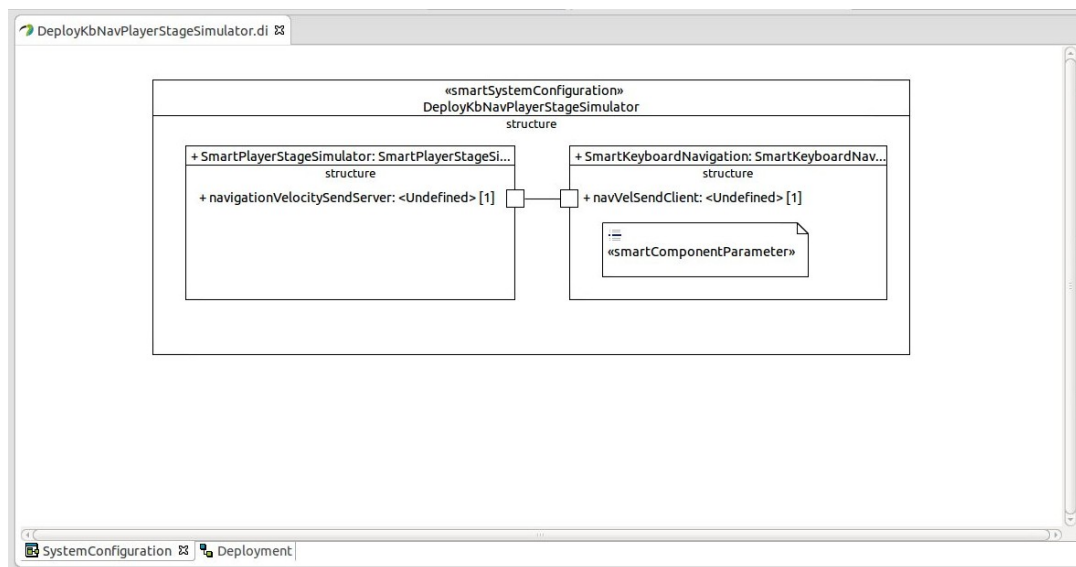


Figure 3.4. System Configuration

Additionally the Deployment has to be modeled. The Deployment contains the following elements:

- SmartDevice (name: PC)
- SmartNamingService (name: NamingService)
- SmartArtifact (name: KeyboardNavigation, utilizedComponentInstance: SmartKeyboardNavigation)
- SmartArtifact (name: PlayerStageSimulator, utilizedComponentInstance: SmartPlayerStageSimulator)

The Deployment is illustrated in figure 3.5.

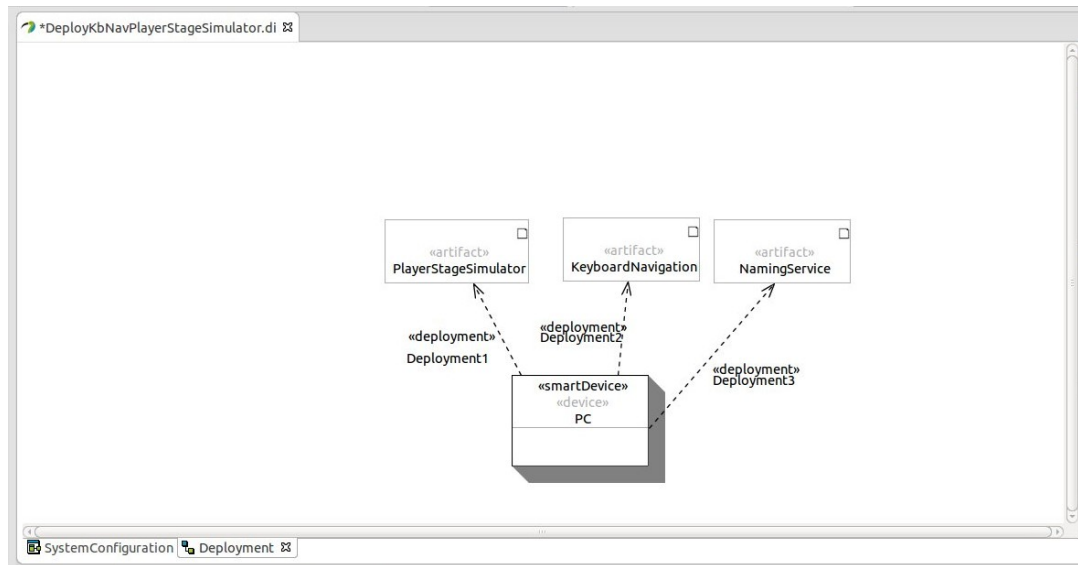


Figure 3.5. Deployment

After generating and deploying the the Deployment project, the scenario can be started. After the scenario is started the world (simulator) can be loaded by typing the corresponding number into the SSH window. The robot can be controlled with the arrow keys or the WASD keys. To do so, the commands have to be typed into the console window of the *SmartKeyboardNavigation* component.

3.2.3.2. System Composition 2: Adding obstacle avoidance

In the previous scenario (section 3.2.3.1) the robot will hit obstacles if the user does not avoid these obstacles in time. To add obstacles avoidance, the following components have to be added to the System Configuration:

- SmartCdlServer
- SmartRobotConsole

Instances of these components have to be added to the System Configuration. Additionally, the services

- "laserServer" of the *SmartPlayerStageSimulator* and
- "navVelSendServer", "navVelSendClient", "laserClient" of the *SmartCdlServer*

have to be connected. The services has to be connected as follows:

- "navVelSendClient" (*SmartKeyboardNavigation*) - "navVelSendServer" (*SmartCdlServer*)
- "laserServer" (*SmartPlayerStageSimulator*) - "laserClient" (*SmartCdlServer*)
- "navigationVelocitySendServer" (*SmartPlayerStageSimulator*) - "navVelSendClient" (*SmartCdlServer*)

Additionally the parameter "plannerInit" of the *SmartCdlServer* component has to be set false, because no path planning is used in this scenario. Furthermore the parameter "dataDir" has to be adjusted, because additional files are necessary for this component. To adjust the parameters of the *SmartCdlServer* component, a *SmartComponentParameter* has to be added to the component:

```

Param server {
  this.plannerInit = false
}
Param cdl{
  this.dataDir = "./"
}

```

The resulting System Configuration is illustrated in figure 3.6.

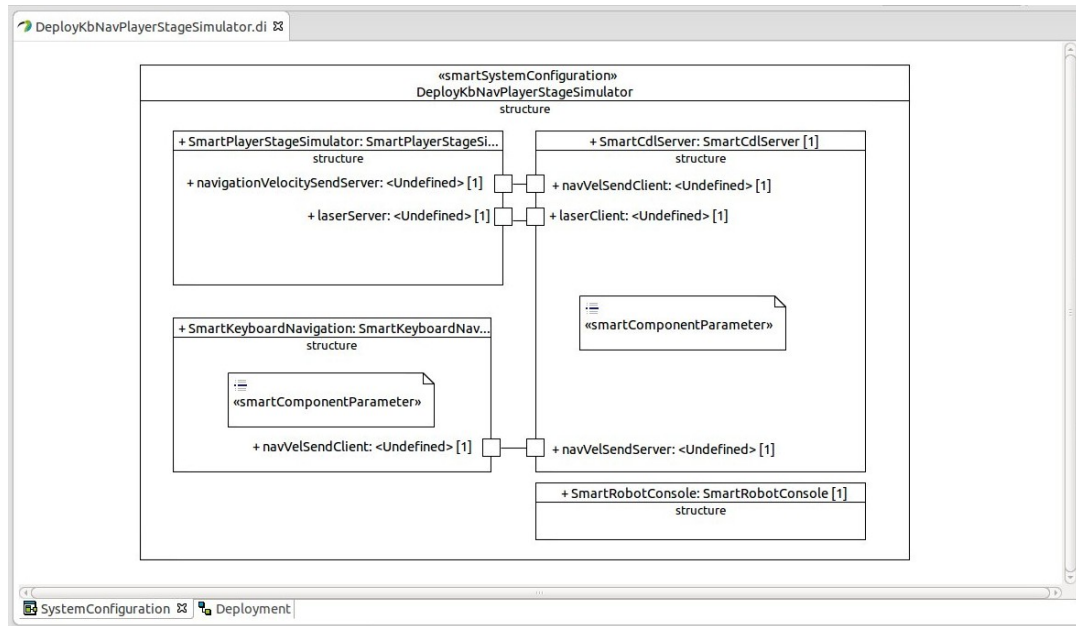


Figure 3.6. System Configuration

In addition the Deployment has to be adapted. Therefore, two additional SmartArtifacts have to be added:

- SmartArtifact (name: CdIserver, utilizedComponentInstance: SmartCdIserver)
- SmartArtifact (name: RobotConsole, utilizedComponentInstance: SmartRobotConsole)

Figure 3.6 shows the resulting Deployment.

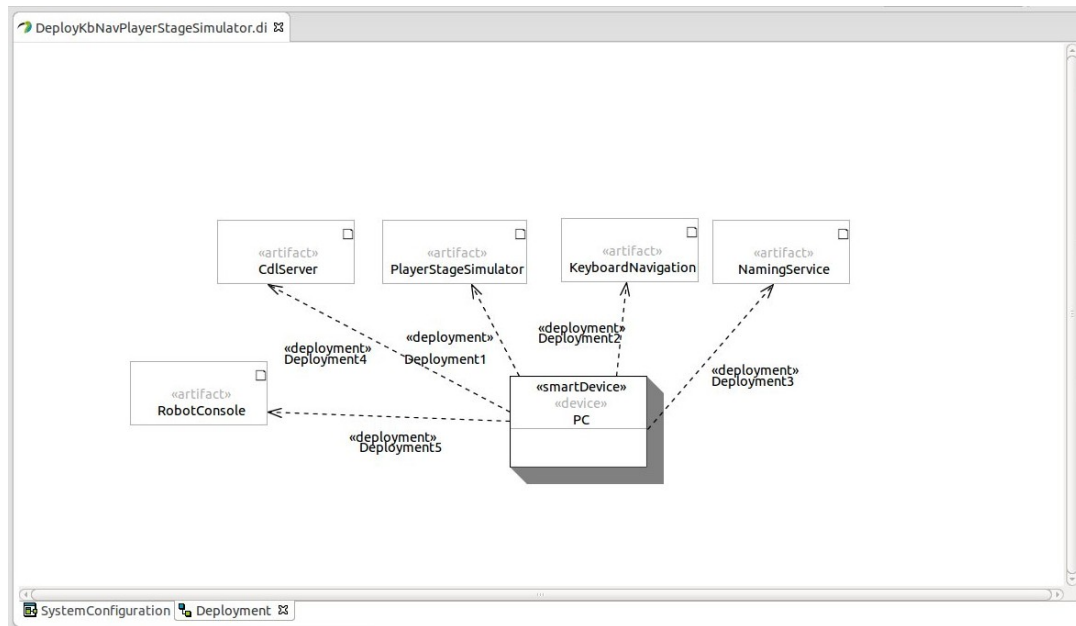


Figure 3.7. Deployment

After the code is generated the files

- CDLacc_P3DX.dat,
- CDLcontour_P3DX.dat,
- CDLdist_P3DX.dat and
- CDLindex_P3DX.dat

have to be copied from \$SMART_ROOT/data/cdl/ to the generated **DeployKbNavPlayerStageSimulator/src/SmartCdIserver_data** folder. These files are required by the SmartCdIserver component (cdl lookup files). Now, the scenario can be started. After choosing a world (simulator) the number 99 has to be typed into the SmartRobotConsole window. To enable the control of the robot with a keyboard "Demo 4" has to be chosen afterwards by typing the number 4. The robot should now be moving controlled by the users keyboard input, however avoiding collisions with obstacles.

Bibliography

These publications can be retrieved online: <http://www.servicerobotik-ulm.de>

- [1] Christian Schlegel, Andreas Steck, Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In Daisuke Chugo and Sho Yokota, editors, Introduction to Modern Robotics. Pages 119-150. iConcept Press, 978-0980733068 (Hard Cover) / 978-1463789428 (Paperback), 2012
- [2] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan F. Inglés-Romero, and Cristina Vicente-Chicote. "Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot", in Journal IT - Information Technology: Methods and Applications of Informatics and Information Technology, Volume 57, Issue 2, Pages 85–98, ISSN (Online) 2196-7032, ISSN (Print) 1611-2776, DOI: 10.1515/itit-2014-1069, DE GRUYTER, March 2015.
- [3] Matthias Lutz, Dennis Stampfer, Alex Lotz, Christian Schlegel. Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns. Workshop Roboter-Kontrollarchitekturen, Informatik 2014, Springer LNI der GI, ISBN 978-3-88579-626-8, Stuttgart, September 2014.
- [4] Dennis Stampfer, Alex Lotz, Matthias Lutz and Christian Schlegel. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". Special Issue on Domain-Specific Languages and Models in Robotics, Journal of Software Engineering for Robotics (JOSER), 2016.
- [5] Christian Schlegel, Dennis Stampfer. "The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development." Tutorial on Managing Software Variability in Robot Control Systems. Robotics: Science and Systems Conference (RSS 2014), Berkeley, CA, July 13th 2014.