IST-002057 **PalCom**

Palpable Computing:

A new perspective on Ambient Computing

PalCom External Report #70
Developer's Companion

Start date of project: 01.01.04
Duration: 4 years

Peter Andersen (ed.), *datpete@daimi.au.dk*
Simon Bo Larsen (ed.), *simonbl@daimi.au.dk*

Revision: 0.1

April 24, 2008

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Integrated Project
Information Society Technologies

Contents

Preface

This document explains the reference implementation of the PalCom Architecture provided by the Open Source PalCom Toolbox. The document has two roles/purposes:

1. A thesaurus/compendium of documentation of the various parts of the PalCom toolbox.
2. A manual/tutorial for people learning about how to make computing palpable by use of the PalCom toolbox.

Section 2 contains a roadmap to use in order to learn to use the toolbox in 3 steps. Using this roadmap a newcomer to PalCom can start out with a pre-compiled demonstration application to learn the very fundamentals of the palpability philosophy, before gradually moving on to more details about constructing and deconstructing assemblies, the programming model, how to use the tools and middleware components and so forth.

1 Introduction

Traditional frameworks for ubiquitous (or pervasive) computing aim at supporting the highly dynamic computing environment where mobile and stationary devices connect and coordinate with means to do so seamlessly and automatically. This is fine in many situations, but if the user is unable to understand and control the system(s) problematic scenarios are likely to occur. That is why we claim that systems should support *palpability*: We do not want to make the computer "invisible", unless we are also able to make it visible if the need for inspecting it arises.

The PalCom Toolbox is a framework for developing computing applications supporting palpability. Using the toolbox and conforming to the programming models it is possible to develop software services with innovative features such as:

- End user understanding of the system
- Possibility of freely combining devices to perform tasks
- Possibility for inspecting the components of the system in levels not anticipated by the developer

The PalCom toolbox provides developers with means for building services that support palpability. It contains a specification and a reference implementation of the PalCom Open Architecture and a collection of services to combine in assemblies, as well as a collection of tools for making this combining easy. The Palcom Toolbox is Open Source and is downloadable under the BSD License.

The PalCom toolbox contains PalCom services and PalCom Components to help developers build palpable applications, but it does not ensure this goal on its own. Palpability happens in *use* of systems allowing the user to obtain understanding and control of the system. Therefore, we find it important for future developers of palpable computing not only to get acquainted to the toolbox described in this document, but also to the theoretical, methodological and conceptual foundations of palpable computing described elsewhere.

2 Roadmap to the PalCom world

This section will try to guide you through a steep learning curve of the PalCom world. Each step will assume that you are familiar with the information given on the preceding steps. We operate with the following steps:

1. **Getting introduced to PalCom**: If you have never before heard about PalCom or palpable computing, this would be where to start. PalCom systems consist of assemblies that are a composition of services and devices. You can learn how to use (and experiment with) assemblies and palpable systems using the pre-compiled demo tools provided on the project web site.
2. **Using the PalCom Frameworks to develop basic PalCom Services and assemblies**: Getting to concepts with the basic PalCom terminology and ideas, this is your level if you are ready to start programming new services using the PalCom Service Framework.
3. **Using the middleware managers and the runtime-environment and changing/extending the PalCom Framework**: The toolbox provides a range of advanced tools on middleware and runtime level for supporting palpability in the developed systems. On other areas it may be necessary to extend the toolbox with new core components core components (e.g. support for new languages, new Media Abstraction Layers, etc.). This companion provide information on a range of issues, and refers to more detailed sources. However, if you can not find the documentation you are looking for on this level, chances are that it was never written anywhere, and you will only have the actual source code to refer to.

After identifying your level of use of the PalCom toolbox, please go to the appropriate section below:

2.1 Getting introduced to PalCom

On the web page, `www.ist-palcom-org/try-it`, you can download pre-compiled demo applications that will allow you to learn and experiment with the assembly concept.

While experimenting with the demo, we suggest that you read section 3.1 for an introduction to the basic PalCom concepts.

If you have already downloaded the toolbox you can also read the tutorial describing how the demo example was build. The tutorials are located in "/doc/tutorials" under the source trunk.

2.2 Developing PalCom Services and Assemblies

Before moving on you need to have the PalCom toolbox downloaded and unpacked to your computer. See section 6 on how to do so.

You also need to make sure you have a few utilities installed:

- Eclipse – see section 6.2.
- Assembly Plug-in – see section 7.2

The very basic concept in a PalCom system is the assemblies that combine the different components of the system into an overall unit providing the required functionality to the user, as it is described in section 4.1.

Reading section 4.2 you can learn about the programming model with the PalCom Service Framework. This enables you to start programming basic software services ("PalCom Services") using the PalCom Service Framework.

At this point it would be in its place to browse through the short version of the PalCom ontology, section 3.2, in order to understand the relations between the various concepts. Knowledge about the Services and Devices in the toolbox is an advantage at this stage, for this you may want to refer to the documentation provided in the Javadoc for the toolbox.

Before you start adding your own services to the code, you should also be acquainted with the file structure in the PalCom toolbox. You can get an overview of this in section 5.

The easiest way to develop a new assembly with your own services is to use the Eclipse-based PalCom Developer's Browser. The user manual for this browser is found in section 7.2.

2.3 Using middleware managers and the runtime-environment and/or extending the framework.

Before moving on you need to check that your system has a few needed external tools installed, see subsection 6.6 of the general section 6 on installation..

You are now ready to learn that PalCom is more than combining devices and services in assemblies. We suggest that you take a look at the remaining issues in the PalCom ontology, which can be found in the PalCom Open Architecture deliverable [7], and that you read the programming model describing how to use the middleware components, see section 4.4.

There is now little left we can offer for you in your way towards PalCom mastery. We realize that the PalCom toolbox is far from complete, and you may want to extend it with new components, for instance a new Media Abstraction Layer, support for a new programming language or other the like.

To do such things you will need a thorough knowledge of the PalCom Toolbox and how the various middleware components glue it all together. The Javadoc API's accessible from the PalCom website would be the first place to look for documentation of these.

And then there is only left to say: Good luck and may the Palpability be with you and your computer systems!

3 PalCom concepts and ontology

This section describes the PalCom concepts on different levels of detail. . A short version of it in section 3.2. For the full version of the ontology, please refer to the PalCom Open Architecture Specification.

3.1 Terminology

To understand the documentation and use of the toolbox it is necessary to get acquainted to the following terms:

Palpability For ubiquitous computing systems to be useful in practice, the user must be able to retain the control, be able to understand the system behavior, and have the possibility to override any policies applied by the system, the system must support *palpability*.

We chose the word 'palpable' because it comes closest to capturing the fact that people need to be able to grasp what technologies *are* doing and *could* do for them if they are to use them effectively and creatively. Palpability is not a fixed property of an object. Instead it arises in interaction as an effect; its shape changes with the changing attention, interests, actions that people bring into different situations. First, to support interaction with computational resources, these resources must be visible to the user. It must be possible to inspect and understand their states. We want this inspection to be generic and its availability not contingent on the ability of application programmers to foresee what needs to be made visible at any given time. Therefore it should be supported in the architecture by a middleware runtime infrastructure. Second, it must be possible to combine resources as prompted by immediate circumstance and take them apart and reconstruct them to explore their workings in detail, and maybe try out different options. This leads to our model of service composition – in the concept of explicitly represented assemblies.

Assembly A palpable computing application is usually made by composing devices and services into *assemblies* that span the functionalities needed for the end user. In this way the assembly becomes the central point of entry to the system presented to the end user.

Assemblies can be constructed, deconstructed and reconstructed either manually or automatically, using scripts; and one important feature of assemlies are their *explicit* representation during runtime.

PalCom Tools A PalCom Tool is a provided utility for browsing, composing, decomposing, inspecting, and/or simulating assemblies.

The PalCom Tools also include utilities for construction of Services and Components, i.e. various compiler scripts.

PalCom Device PalCom Devices are hardware entities (or simulated hardware entities) capable of hosting PalCom Services, i.e. either executing the PalCom runtime environment or implementing the communication protocols needed to do so.

PalCom Service A PalCom service is defined as a limited functionality hosted on a device and made available to the user or to other systems through interfaces.

3.2 The short PalCom ontology

To help understand all terms and concepts used in the description of the PalCom toolbox, figure 1 depicts the concepts and their logical relations.

The nature of the software systems in PalCom is that they consist of small subsystems that are self-contained and communicate with each other to solve the tasks needed by the users. Therefore, the choice of architectural style gave itself as a service-oriented architecture (SOA). A SOA is a collection of services that communicate with each other. The services are self-contained and do not depend on the context or state of other services. They work within distributed systems architectures.

This separation allows services to be distributed to different machines and different locations. At the same time it offers the possibilities for the user to bring together any services in assemblies with explicit representations.

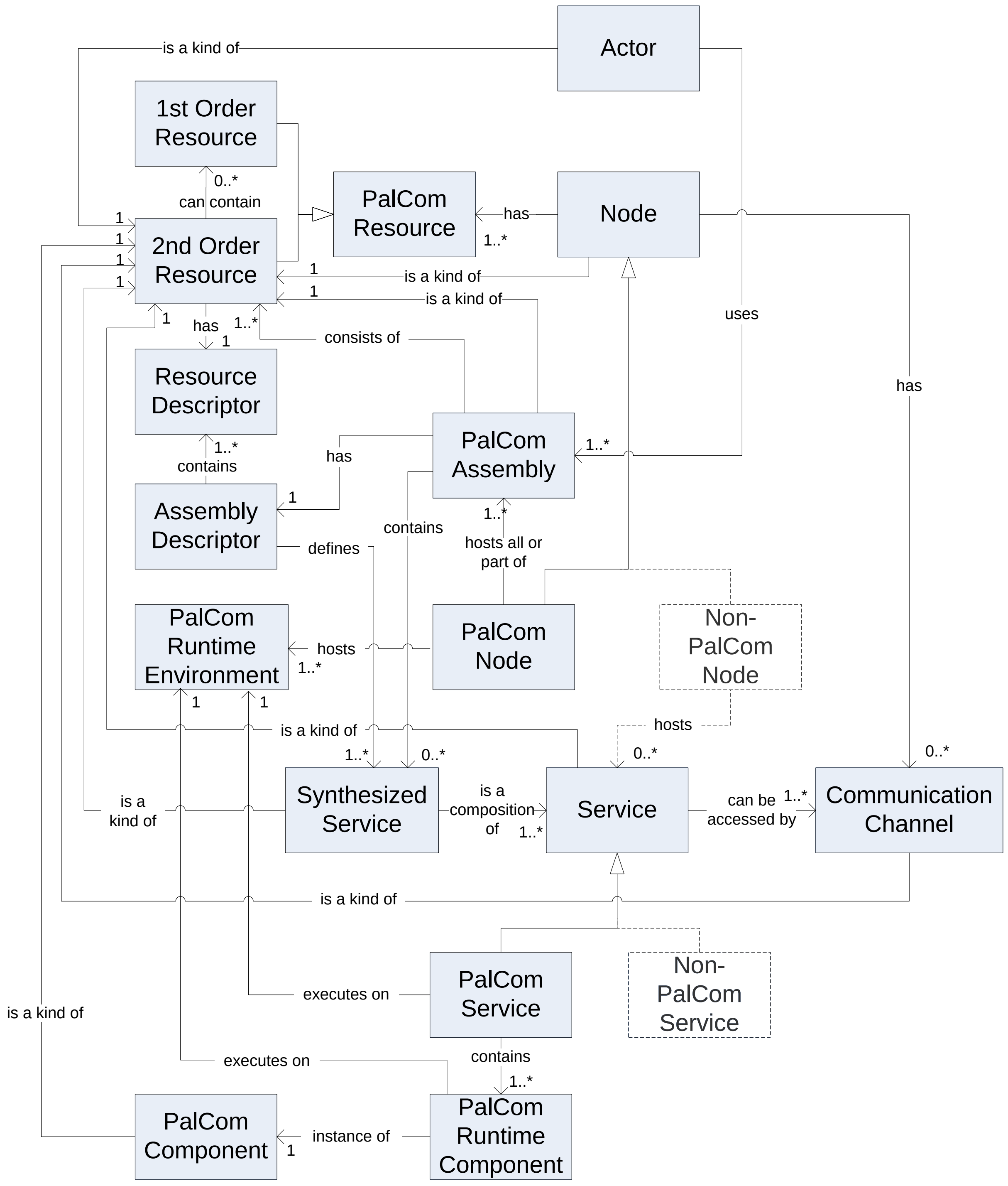


Figure 1: Overview of the palcom ontology

What sets assemblies apart from other composition mechanisms is that an assembly more or less is represented explicitly in the software. This means there is something that can be subject of inspection rather than just setting up connections between individual services.

Inspection is extremely important, for example, if users should be able to deconstruct an existing assembly in a sensible manner. To support e.g. sharing of assemblies among colleagues, it demands an explicit representation of the services combined as a whole.

The vision in PalCom is to develop a more understandable, easy-to-use mechanism for constructing, de-constructing, and re-constructing when, for example, one is reusing parts of other systems in unforeseen new combinations by using this assembly mechanism and managing contingencies.

3.3 The full PalCom Ontology

To see all details of the ontology, please refer to the description of the PalCom Open Architecture, `http://www.ist-palcom.org/publications/deliverables/Deliverable-54-[2.2.3]-open-architecture.pdf`

4 PalCom Programming Models

Using the PalCom toolbox can be done in different ways and with different levels of knowledge of the 'inside' of the toolbox as also explained in the roadmap to the documentation, see section 2.

The top-level of a palpable system is the assembly binding together the devices and services. Thus, the first programming model to understand is the assembly model, section 4.1.

The toolbox comes with implemented services to make the assemblies used in the PalCom application prototypes, but in most cases these will be insufficient to cover other purposes. Therefore it is necessary to develop your own palcom services.

This documentation will focus on developing palcom services in Java. There is a service framework provided in the toolbox for this case, and section 4.2 describe the programming model for using this Java framework.

However, one of the strengths of the PalCom Open Architecture is the further platform and language independence obtained by providing the PalCom Virtual Machine (`pal-vm`) as an alternative to running on the standard Java Virtual Machine (JVM). Section 4.4 describes the steps involved with programming for running on the `pal-vm`.

4.1 The PalCom Assembly model

As written in section 3.1, a PalCom Assembly is the primary entry point between a user and a PalCom system.

Ontologically, the assembly consists of an organized collection of 2nd Order Resources (see section 3.2) such as services or devices.

An assembly exists at runtime in the form of an assembly manager which manages the runtime execution as specified by an assembly descriptor. At development time the assembly is structured using an assembly descriptor. The assembly descriptor can be written manually, but are more conveniently created using the assembly browser described in section 4.1.1.

In addition to the composition template, the assembly's assembly descriptor specifies the behavior of the assembly when it encounters various forms for events.

In some cases it is desirable for the assembly to be part of another assembly. This is achieved by exposing the assembly as a Synthesized Service that can be incorporated by the other assembly. It is recommended, but not mandated, that an assembly define only one Synthesized Service in order to restrict complexity.

An assembly may be distributed over multiple physical devices and if so will include the definitions of Communication Channels required to connect the remote parts.

Just like object orientation provides not only programmatic mechanisms but also an accompanying view of how to design systems, the assembly concept has certain implications for how palpable systems are developed. In particular, the key implications for the programming model are:

- The assembly construct does not rely on separating runtime from development time. It would not be feasible to do so because the dynamic habitat of services in which an assembly exists cannot be 'shut down' and re-compiled. Thus runtime binding and change of the assembly is a must. Further, assemblies themselves also need to be dynamically reconfigurable because it is the central construct with which the architecture enables end-user programming. Therefore it is important to enable users to mix tailoring of an assembly with its use.
- It encourages developers to structure systems in a way that is easier to understand for users. In palpable computing assemblies replace the traditional notion of an application. Thus to the users it is the entry point for understanding and inspecting a system they are using. As an architectural construct designed to support both the user and developer perspective on a system, the programmers and developers who build systems are forced to so using constructs that furthers understandability to users.

Figure 2 shows an assembly descriptor for a simple assembly with two services.

The assembly descriptor describes which devices, services and connections exist in the assembly, as well as an optional EventHandler script which can be used to specify the behaviour and interface the synthesized service in case the assembly has one.

In the example there are two services both located on the same device, and with a connection between them. Even in an assembly as simple as this one it is useful to use the assembly model rather than just connecting the two services independently of any assembly.

There is two basic ways to construct (or modify) assemblies: Manually using an assembly browser or automatically using an assembling script.

4.1.1 Assembly browsers

In the PalCom toolbox, two different assembly browsers are provided. The first on these is a general overview browser for instance useful to inspect running assemblies. The second is made as a plug-in to the Eclipse development environment and is first of all thought of as a developer tool for convenient usage of assemblies alongside the development process. An exported "stand alone" version of this second browser is also available, and can be launched with the `pal-developers-browser` script, see section 7.2 for details.

Overview Browser The Overview Browser (OVB) provides a "window" into the PalCom environment, where deployed devices and services on the LAN and their connections in assemblies are shown.

The OVB can be started out-of-the-box with the `pal-ovb` and `pal-dev-ovb` scripts. A user guide for the OVB is included in section 7.1.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE PRDAssemblyD SYSTEM "palcom.dtd">
<PRDAssemblyD format="2" name="MyExample">
  <VP id="174682">
    <DID id="C:d0f40e01-10a3-408b-bdb1-042a735f69be" />
  </VP>
  <PRDAssemblyVer format="2" name="MyExample" released="false">
    <AID cdid="C:d0f40e01-10a3-408b-bdb1-042a735f69be" cn="174682"
         udid="C:d0f40e01-10a3-408b-bdb1-042a735f69be" un="174682" logVer="1.0" />

    <DeviceDeclList>
      <DeviceDecl>
        <Identifier id="MyDevice" />
        <DID id="C:1f654c57-04ab-4d0e-9eb2-524f6a1da482" />
      </DeviceDecl>
    </DeviceDeclList>

    <ServiceDeclList>
      <ServiceDecl>
        <Identifier id="sender" />
        <SingleServiceDecl>
          <Identifier id="sender" />
          <DeviceUse>
            <Identifier id="MyDevice" />
          </DeviceUse>
          <SIID in="1">
            <SID cdid="X:1" cn="TF35" udid="X:1" un="TF35" />
          </SIID>
        </SingleServiceDecl>
      </ServiceDecl>
      <ServiceDecl>
        <Identifier id="receiver" />
        <SingleServiceDecl>
          <Identifier id="receiver" />
          <DeviceUse>
            <Identifier id="MyDevice" />
          </DeviceUse>
          <SIID in="1">
            <SID cdid="X:1" cn="TF39" udid="X:1" un="TF39" />
          </SIID>
        </SingleServiceDecl>
      </ServiceDecl>
    </ServiceDeclList>

    <ConnectionDeclList>
      <ConnectionDecl>
        <ServiceUse>
          <Identifier id="sender" />
        </ServiceUse>
        <ServiceUse>
          <Identifier id="receiver" />
        </ServiceUse>
      </ConnectionDecl>
    </ConnectionDeclList>

    <EventHandlerScript>
    </EventHandlerScript>
  </PRDAssemblyVer>
</PRDAssemblyD>
```

Figure 2: Assembly Descriptor Example. For editing Assembly Descriptors, we recommend using the Eclipse-based Developer's Browser, see Section 4.1.1

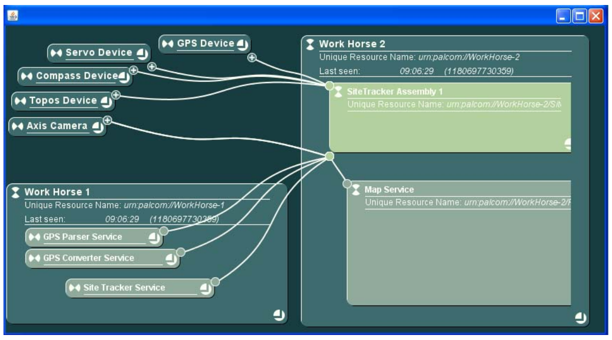

GPS Device
Servo Device
Compass Device
Topos Device
Axis Camera
Work Horse 2
Unique Resource Name: urn:palcom://WorkHorse-2
Last seen: 09:06:29 (1180697730359)
SiteTracker Assembly 1
Unique Resource Name: urn:palcom://WorkHorse-2/S
Map Service
Unique Resource Name: urn:palcom://WorkHorse-2/
Work Horse 1
Unique Resource Name: urn:palcom://WorkHorse-1
Last seen: 09:06:29 (1180697730359)
GPS Parser Service
GPS Converter Service
Site Tracker Service

Figure 3: Screen shot of the Overview Browser showing the SiteTracker assembly

PalCom Developer's Browser See Section 7.2 for a description and user's guide to the PalCom Developer's Browser.

4.2 PalCom Service Framework programming guide

To ease the programmer's task of creating new services, a service framework is provided. It contains the classes are *AbstractDevice* and *AbstractService* and various classes supporting communication with other services, including service discovery. More details of the classes in the service framework can be found in the Javadoc under packages `ist.palcom.services` and `ist.palcom.device`. The echo device (package `ist.palcom.services.echo`) has been thoroughly documented to provide an example of how the service and device frameworks can be applied. It can be feasible for a newcomer to PalCom to have the echo device source code in hand while reading the following programming guide.

AbstractDevice AbstractDevice can be subclassed for creating a new PalCom device. It provides subclasses easy access to most basics of a device, including communication, configuration and logging. It holds the services and managers on the device, and it has a PalcomScheduler that schedules and runs the services and managers. Services inheriting from AbstractService can be attached and are hereafter immediately discoverable and accessible.

AbstractService AbstractService is the superclass of all services written in the framework. It has name and address information about the service.

Programming PalCom Services with the service framework is essentially a matter of subclassing the *AbstractDevice* and *AbstractService* and using the API of the communication classes. However, for a newcomer there may also be some practical issues that needs explanation, even though it may seem trivial. These issues are described in the following.

There are multiple ways to go when programming for PalCom, but the one described here seems to be the most straight forward for newcomers. It is assuming that the developer uses Eclipse (version 3.3 recommended).

Practicalities in getting started Before creating anything, it is of course necessary to have the PalCom toolbox source code checked out (see section 6).

The build process of the source code is described in more detail in section 6.4, but a simple step-by-step guide is provided here:

1. Open eclipse, create a new workspace, disable automatic build (under "Project" in the menu bar), set JRE and compiler (Window->Preferences->Java) to 5.0 or newer.
2. Import all the PalCom projects intro the workspace: Select Import->Existing Projects into Workspace. Set root directory to your palcom checkout directory. Select all projects and click Finish.
3. Compile the projects, using the "build.xml" ant script in the dev-lib project as described in section 6.4.

Creating a new PalCom Service The following step-by-step guide explains how to get started and initialize a new PalCom Service. As before these steps only represent one way of doing it.

1. Create a new project: File->New->Java Project. Name the project and make sure there is a source directory in the project (the eclipse "Java Project" wizard does this automatically).
2. Create new package for your service: Right-click on the source folder and select New->Package.
3. To be able to compile your package, you must set the build path to include the `ist.palcom.services` package. Right-click the package just created and select Build Path->Configure Build Path. Add the services package under the "Projects" pane.
4. Create two classes: One service and one device (i.e. One class inheritting from abstractService and one from AbstractDevice).
5. Add constructors to both classes. The constructors should call the constructors of the super class, i.e. for the device: `super(deviceID);`, and the service:

   ```
   super(context, serviceName, true, false, PRDService.UNICAST, "P1", false, "1",
         readableName, createServiceProxy(), localSID);:.
   ```

6. Implement the initDevice() method in the device, and make sure it calls called `super.initDevice()` and instantiates and starts the service(s). The service(s) should be kept as private variables in the device. The first parameter in the service(s)' constructor is the device context holding various info for the device, and the initDevice() method must also make sure the service(s) are registered in the context object.
7. Define the "static" serviceproxy in the service by implementing the `createServiceProxy()` method. The service proxy is called so because it serves as kind of interface for the service, specifying which commands are available for interaction with the service.
8. Create the main method in the Device class. The method must instantiate the Device, e.g. using a custom hard coded DeviceID and device name.

Following these steps you should now have an "empty" device/service implementation - i.e. a device with service(s) that does not do anything. You can check that it works by running the device and see it appear in the overview browser or the developer browser (See section 4.1.1).

If it works, we are ready to make the actual functionality of the service.

Programming PalCom Services Functionality in PalCom Service will usually be implemented in threads allowing for concurrent execution in case multiple assemblies connect to the same service.

One typical way of designing the service is to make one thread for each command in the service. PalCom threads are made as inner classes in the service extending the PalComThread (or the PalComPriorityThread):

Communication between threads happens with `Event`'s or `CommandEvent`'s in case they are triggered by one of the service's commands, e.g. by a user.

1. Declare the inner class extending PalComPriorityThread. Implement the `run()` method of the thread using a `while(true)` loop for encircling the event handling.
2. First line in the loop should wait for an event to occur, using the `waitEvent();` method. The event handling logic is then implemented. Based on e.g. the `instanceof` the incoming event, appropriate action can be taken for the service's commands (that can be accessed via the service proxy – see above).
3. Appropriator events for this particular service can be declared as another inner class in the service extending the `Event` class. This could for instance be useful if communication between two threads in the service is needed.
4. Finally, `start()` and `stop()` are implemented in the service. `start()` is responsible for calling `super.start()` and for registering the threads in the scheduler of the device context. `stop()` is used for terminating the threads and calling `super.stop()`.

It is highly recommended that services is given means to be inspectable. Section 4.3 describes how to incorporate use of HGraphs in the service.

Out-of-the-box usage It is also possible to create PalCom services without checking out the PalCom source code, relying instead on out-of-the-box usage of the PalCom precompiled jar files. For more information on this refer to section 4.4.3.

More information In /palcom/doc/tutorials/frameworks/device-service.html the usage of the service framework is described in further detail using the EchoService as an example. The example contain code presenting a result of following the above steps.

More info on service interaction can also be found in section 7.1.4 in the Open Architecture Specification, [7].

Finally, the Javadoc documentation of the device- and service frameworks contain APIs for usage of the various classes. See the packages`ist.palcom.services` and `ist.palcom.device` from the page `http://svn.ist-palcom.org/svn/palcom/trunk/doc/javadoc/html/index.html`

4.3 Introspection with H-Graphs

As discussed in the conceptual framework deliverable[6], reflection plays an important role in making systems palpable. A number of tools and architectural facilities are used to enable user inspection of a program. Here we deal mainly with the reflective features of our infrastructure that facilitate such inspection.

When writing a program, it is possible to do so in a way that makes any part of its state accessible and modfiable from the outside.

Suppose a component wanted to provide access to one of its internal variables X for its clients to read and write. This could easily be done by equipping that component with an interface containing $readX()$ and $writeX(...)$ methods. Why, then, is reflection useful ? The fundamental feature of reflection is genericity. It allows inspection of a program irrespective of the programmer's ability to anticipate what may need to be made externally accessible.

The basic approach in PalCom is to use a Hierachical graph, or H-Graph, to achieve the reflective abilities of the software. Actually, the H-Graph construct spans a complete and new programming model for making PalCom systems, but for now we will simply focus on how to make exisiting services and their internal state inspectable. (For more information on the broader use of HGraphs see section 7.3. in [7] and/or Working Note #115.)

To use a data structure inspectable it must be mounted in the HGraph by one of two ways:

1. It must implement the `IHgraphNode` interface, see section 4.3.1.
2. It must be defined as a Value object on an existing H-graph node, see section 4.3.2.

4.3.1 Nodes

Creating In order to create a new H-Graph node, one has the choice between either extending the `ist.palcom.hgraph.HgraphNode` class or implementing the `ist.palcom.hgraph.IHgraphNode` interface. In implementing the `IHgraphNode` interface, it is recommended that the implementation is based on an internal private `HgraphNode` as with e.g. `ist.palcom.hgraph.manager.HgraphManager`, only overriding actual functionality intended to be overridden. Given that H-Graph nodes are required to have names, it is the responsibility of the implementer to ensure that all classes implementing the `IHgraphNode` interface have a name.

Instantiating The `ist.palcom.hgraph.HgraphNode` is instantiated using either `new HgraphNode()` or `new HgraphNode(String)`, where String is the intended name for the node. Given that HGraph nodes are required to be named, the default constructor sees to it, that the node is assigned a name. Within the current implementation of `HgraphNode`, this is accomplished using the following naming scheme: `"ANONYMOUS" + super.toString()`

Adding Nodes are added with `int addNodeChild(IHgraphNode)`, which returns `IHgraphNode.STATUS_OK` if the operation succeeds. If a sibling with the same name exists, the method returns `IHgraphNode.CHILD_CIRCULAR` Being added to an H-Graph triggers an automatic recalculation the path of the node and its offspring.

Removing Nodes are removed with `int removeNodeChild(String)`, where String is the name of the node to be removed. When removing a node, relevant sibling references are updated by nullifying the actual reference (see below for more). The method returns `IHgraphNode.STATUS_OK` if the operation succeeds.

Renaming An `IHgraphNode` can be renamed using `int setName(String)` where String is the new name. If the node is installed in an H-Graph, it is checked whether there already exists a sibling with the given name. If so, an error code is returned. Further, if there are sibling references delaing with the given node, these are updated accordingly. Lastly, the path of the node and subsequently its offspring is recalcluated.

4.3.2 Values

Instantiating A new Value is instantiated using `new Value(String, Object)`, where the String is the name of the value and the Object is the encapsulated *value* of the Value.

Adding Values are added to a given H-Hgrap node, using `int addValue(Value)`. The method returns `IHgraphNode.STATUS_OK` if the operation succeeds. If a Value with the given name already exists, the value is not added and the method returns `HgraphNode.VALUE_ALREADY_EXISTS`.

Changing Object values contained within Values are changed using `int setValue(String, Object)`, where the String is the name of the Value which is to be changed and the Object is the new value. If no Value exists with the given name, a new one is created and added.

Removing Values are removed using `int removeValue(String)` where the String is the name of the Value which is to be removed. The method returns `IHgraphNode.STATUS_OK` if the operation succeeds or if no Value with that given name is found.

Renaming There is, at this point, no direct way of renaming Values. This is as such accomplished by attaining the value of the original Value, removing that and adding a new Value with the new name. This is completely unproblematic given that the loose coupling between the Values and the users of the values is upheld.

4.4 Dual VM Programming Model

The general programming model for PalCom described section 4.2 contained specific examples based on the `pal-vm` implementation of the PalCom Runtime Environment. However, for the more advanced user of the toolbox (e.g. a PalCom Component developer) there is need for some more details to the "programmer's view" of the `pal-vm` implementation. These will be given in this section.

4.4.1 Dual Execution Engine

Chapter 4 of the PalCom Open Architecture deliverable [7] contains a figure showing the layered architecture of a given PalCom complient device. The first entity in the layer above the Execution Engine layer is named "Runtime Engine" in that figure. As already mentioned in the introduction to this chapter, the implementation of the PalCom architecture in the open source code base contains a dual execution model. This is illustrated in Figure 4, which details the Runtime Engine entity.

The essence is, that on most machines, one can choose to execute PalCom code on either the Java Virtual Machine (JVM) or the PalCom Virtual Machine (`pal-vm`). The Core Libraries, Middleware Management, and 2nd Order Resources are all written in Java code, compatible with `pal-j`, which means that they can be compiled with either the standard Java Compiler `javac` and executed on JVM, or with `pal-j` and executed on the `pal-vm`. See Section 4.4.4 below for details.

4.4.2 `Pal-vm` Out-of-the-box or Developer Use

As can be seen from the code base described in section 5.1, there are two different ways to work with the `pal-vm`: The developer can either use the "out-of-the-box" precompiled binaries and libraries in `palcom/bin` and `palcom/lib`, i.e., `pal-vm`, `pal-st`, and `pal-j`. Working this way requires no other preparation than checking out or downloading the code base [4], and setting the PATH to include `palcom/bin`.

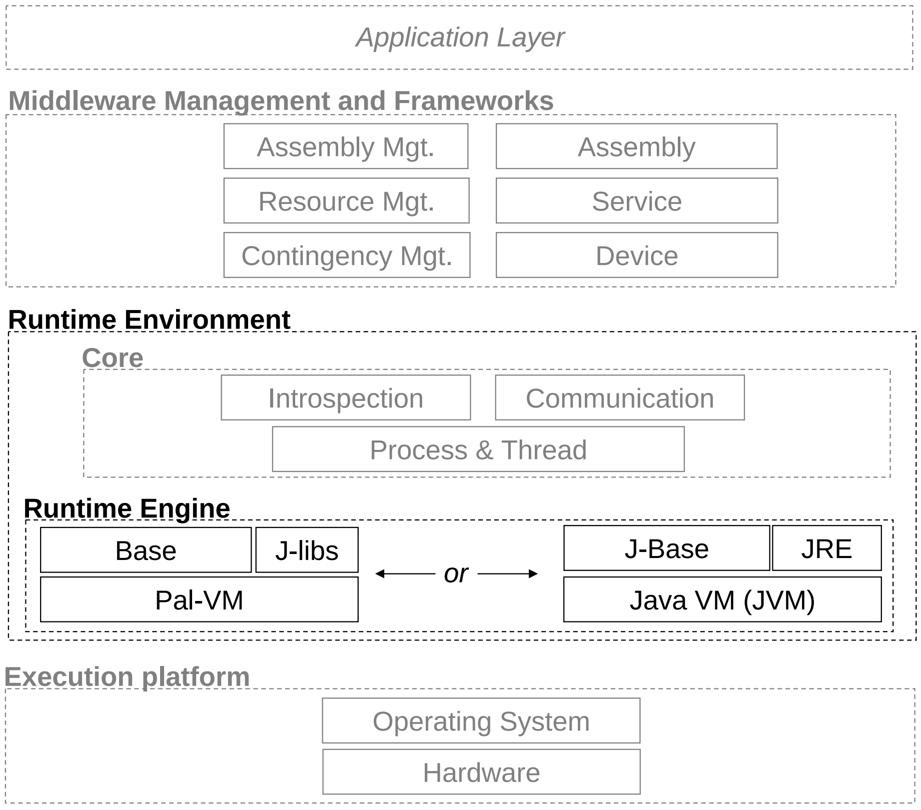


Figure 4: Runtime Layers on one PalCom Device

Alternatively, the developer may work with the developer tree included in the PalCom SVN. This means that s/he will be using the `pal-dev-*` scripts included in `palcom/developer/dev-bin` (e.g., `pal-dev-vm`, `pal-dev-st`, `pal-dev-j`), which use the latest developer version of the tools and libraries in the SVN tree. Doing so requires the developer to have `palcom/developer/dev-bin` in the PATH, and have executed the script `pal-dev-build`.

Unless otherwise noted, the programming cycle described below assumes "out-of-the-box" use. For "developer" use, simply use the `pal-dev-*` scripts instead.

4.4.3 JVM Out-of-the-box or Developer Use

Also for the JVM, there are both "out-of-the-box" and "developer" uses of the toolbox. For the former, it is enough to put one of two JAR files in `palcom/libs/jars` on the compile-time CLASSPATH: `palcom.jar` or `palcom-applications.jar`. With `palcom.jar`, framework classes for building devices and services are available. `palcom-applications.jar` also provides a number of example services, in addition to the framework classes.

As for the `pal-vm`,"developer" use means to work against the latest SVN tree. This is especially useful when making modifications to the toolbox, but it may also simply give a better overview of the toolbox contents. For Eclipse, there are a number of projects in the tree that can be imported and put on the build path of new projects. When working against the SVN tree outside Eclipse, there are two generated JAR files under `palcom/developer/dev-lib/jars/generated`, corresponding to the ones mentioned above, which can be put on the CLASSPATH. These JAR files are generated automatically by the build process in eclipse, and by `pal-dev-build -J` when working on the command line.

4.4.4 Programming Cycle

The typical programming cycle when working with the PalCom toolbox will start with Java programming, e.g. in Eclipse, and execution of services and assemblies from either inside Eclipse or using JVM from the command line. By using the `pal-vm` tool suite 7.5, one may extend this programming cycle to include deploying to and execution on devices supported by `pal-vm`.

A different programming approach can be taken by programming in Smalltalk, and compiling it with the `pal-st` compiler. Using this approach, the developer will typically skip the use of, e.g., Eclipse completely. They will still be able to utilise the Core, Infrastructure, etc. libraries (written in Java), through the use of the language interoperability mechanisms described in the PalCom Open Architecture deliverable [7]. But often, the program will directly use the (Smalltalk based) methods found in `ist.palcom.base`. Doing this will result in the fastest and smallest code, compared to doing it in Java, but the resulting program will, of course, only be able to run this on the `pal-vm`, and not the JVM.

Figure 5 gives an overview of the various paths one may follow in this programming process by showing the various file types used in the toolbox and how they are connected with the various tools.

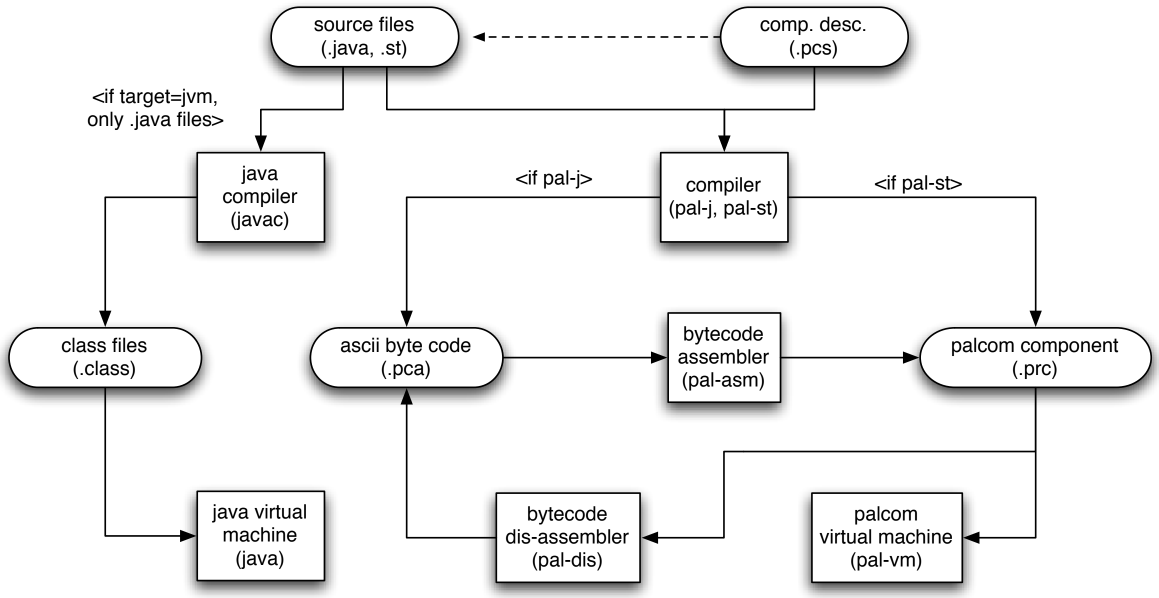


Figure 5: Diagram of the file types and the compile/assembling tools in the PalCom toolbox.

The file types and tools shown in figure 5 will all be described in the following.

Developing in Java using Eclipse

For a Java programmer, a programming cycle would typically start with initial programming on a standard Java Desktop installation, perhaps using the open source Eclipse development environment [2]. The PalCom Repository contains a number of directories containing Eclipse projects in the form of `.project` files and supporting stuff, e.g. ANT scripts with CLASSPATHs set up appropriately for JVM execution. The programmer can then run their program from within Eclipse, and thus execute on the JVM. If the programmer wants to deploy the code (typically services developed using the Service framework), s/he will use the Eclipse export-to-jar feature, and deplay the JAR file(s) thus produced.

Compiling Components for the `Pal-vm`

The next step for the Java programmer would then be to recompile the source code using the `pal-j` compiler (on the command line), and then execute the resulting component(s) on the `pal-vm`, cf. below.

Whereas the unit of deployment for JVM typically are JAR files (corresponding to Java packages), the binary deployment units that `pal-vm` can execute, are so-called *PalCom Component Files*, which are binary files with extension `.prc`.

A *Palcom Component Specification* (PCS) file is used as input to the `pal-j` and the `pal-st`. The PCS file contains the list of classes to be compiled into the binary component, the name of the main class, and a list of components required by the component being specified. [1]

Examples of use can be found in the code base at `http://svn.ist-palcom.org/svn/palcom/trunk/doc/tutorials`.

One such example `http://svn.ist-palcom.org/svn/palcom/trunk/doc/tutorials/interop/basename/src/jbank/jbank.pcs` is:

```
component jbank {
  class JBank
  class JHistoryAccount
  mainclass JBank
  requires "stbankaccount.prc" as stbankaccount
  requires "java.lang.prc" as java.lang
}
```

This file specifies a component named, `jbank`, containing the classes `JBank` and `JHistoryAccount`. The `jbank` component requires the Smalltalk component `stbankaccount.prc` and the standard Java component `java.lang.prc`. The `mainclass` declaration specifies the class to execute an instance of, when creating a process using this component. The `mainclass` declaration is optional for Java, since the `pal-j` compiler can infer the main class automatically by finding a class with a `static void main` method. The `requires` declarations assign a local name to the required components, which is the name used for these components in the scope of the `jbank` component.

PCS files used as input to `pal-j` contain only classes described in Java source files and the `.java` extension is automatically added to the class names. So in this example, compilation of `jbank.pcs` will automatically compile the files `JBank.java` and `JHistoryAccount.java`. Similarly, PCS files for the `pal-st` compiler reference only classes written in Smalltalk. All PCS files can reference components generated by any compiler (`pal-j` or `pal-st`).

The complete syntax for PCS files is defined in Appendix A.

A tutorial on how to program with `pal-j` is available in the PalCom Repository at `http://svn.ist-palcom.org/svn/palcom/trunk/doc/tutorials/pal-j`.

[1] A source language for the `pal-vm` may have its own constructs that are more natural for describing components than the component specification files, and choose to use these constructs instead. In future versions of `pal-j`, the use of component specification files may be replaced by constructs from the Java programming language. The Smalltalk language does not have any construct for describing components, and therefore the component specification files will continue to be used for `pal-st`

Running on a Desktop `Pal-vm`

When ready, the program can then be executed on the `pal-vm`, typically first on the Desktop machine. Concretely, as mentioned above, the outcome of the `pal-j` or `pal-st` compilation(s) will be one or more PalCom Component Files. Assuming these are placed in the current directory, the execution using `pal-vm` is done from the command-line using the command

```
> pal-vm -cp . mycomponent.prc
```

The `-cp .` argument instructs the VM to add the current directory to the *component path* it uses to find components. The `pal-vm` in this command line is one of the scripts explained in section 7.5 and it automatically adds `palcom/lib` to the component path. This is the location of the precompiled components for the standard palcom libraries - i.e. the implementations of the different entities in figure 4. If the developer is using the "developer" approach, the `pal-dev-build` script generates components into the default component path `palcom/developer/dev-lib`, which is automatically added to the component path by the `pal-dev-vm` script.

Deploying `Pal-vm` and Components to a non-desktop Device

Once the code is running on `pal-vm` on a Desktop machine, the developer may want to deploy the components (including necessary system components from `palcom/lib` or `palcom/developer/dev-lib`) to the target device.

The first step in doing this is to identify which components to copy to the target device. To aid in this process, a special tool `pal-dep` has been created (`pal-dev-dep` if using the developer tree). To use this, simply call it on the command line, with the component specified as the main component as argument, e.g. (output paths cropped for readability):

```
> $ pal-dev-dep -l jbank.prc
.../palcom/developer/dev-lib/java.lang.prc
.../palcom/developer/dev-lib/ist.palcom.base.prc
stbankaccount.prc
.../palcom/developer/dev-lib/java.io.prc
.../palcom/developer/dev-lib/ist.palcom.j.runtime.prc
jbank.prc
```

The `-l` options makes the output be separated by newline characters.

Furthermore, the developer will need to build and copy the binary `pal-vm` for that platform. This is done by

1. Invoking the script `pal-dev-build-vm` with a cross build option for the target platform. E.g., to build the VM for UNC20, the command is `pal-dev-build-vm -t unc20`. The available options are displayed by executing `pal-dev-build-vm -h`.
2. Copying the binary VM to the device. The `pal-dev-build` script reports where the binary to copy is placed.

GZIP compressed Components

To allow for deployment onto smaller devices, the component files may be compressed with the standard `gzip` utility. The `pal-vm` has been modified to be able to open such compressed component files directly. Furthermore the `pal-asm` and `pal-st` bytecode assembler has been augmented with a `-z` option, which makes it produce such compressed component files. The `pal-j` compiler produces textual `.pra` assembler files, so to get a compressed component from Java, use `pal-asm -z` on the generated assembler file.

4.4.5 JVM or `Pal-vm`?

Depending on thedeveloper, the coding cycle described above may be more or less appropriate.

Using PalCom on the JVM provides (among others) the following advantages:

1. Full-featured programming environments like Eclipse, with powerful source level debugging
2. The application may make use all the powerful standard libraries, including advanced graphics

Using PalCom on `pal-vm`, on the other hand, at the present time provides the following advantages:

1. Smaller devices, not just Desktop machines.
2. Support for resource monitoring
3. Dynamic languages (currently Smalltalk) as well as strongly typed (currently Java)
4. Language interoperability
5. Isolated processes on one VM
6. Efficient lightweight scheduling in the form of Coroutines

If the application is only to run on the Desktop platform and does not have the need for the `pal-vm` specific primitives, or has specific needs, e.g., advanced graphics, then the developer may want to (or need to) only code for the JVM. We recommend that development is done in Eclipse.

If, on the other hand, the application need the special things provided by `pal-vm`, it is, of course, optional whether the developer wants to code it in Java in Eclipse first. As mentioned above, one benefit from doing this, is that the developer currently gets much better debugging support when running on the JVM, than when running on `pal-vm`.

4.4.6 Language Restrictions

Even though Figure 4 indicates that if one programs in Java, one can freely choose between running on the JVM and `pal-vm`, this is not unrestrictedly the case. Firstly, as described in the PalCom Open Architecture deliverable [7], the entire Java language is not supported by the `pal-j` compiler, so if one uses, e.g., floating point numbers in the code running on JVM, this will not function with `pal-j/pal-vm`. Furthermore, the full set of Standard Java libraries are not supported on `pal-vm` – see the following section – whereas some specific PalCom primitives are only available on `pal-vm`, e.g. resource monitoring primitives.

4.4.7 Base Libraries

In Figure 4 the libraries depicted right above `pal-vm` and JVM, respectively, need a little extra explanation: As noted, all libraries above the Runtime Engine layer are currently written in Java, allowing for reuse on the two execution paths through the layers. As also explained in Section 5.1, the Base library used by `pal-vm` (in Section 5.1 referred to as `pal-base`), contains the basic classes like `Object`, `Integer`, `Boolean`, etc., but also slightly more complex classes like `CheckedArray`, `Process` and the `System` class.

When programming for the JVM, corresponding Java classes are available in the J-Base libraries (`pal-jbase`). Analogue to this, a small subset of the standard Java libraries (called JRE in the figure) is available when running on the `pal-vm` in form of the `j-libs` library (in a mixture of Smalltalk and Pal-J code). When coding in Eclipse, the CLASSPATHs of the projects are actually set up in such a way, that Eclipse checks application code against the J-Libs libraries there. This prevents use of JRE classes not supported on `pal-vm`.

4.4.8 Components and Interfaces

Below it is explained what the `pal-vm` implementation of the PalCom component model is described means from a programmer's perspective:

As already explained above, the `pal-vm`, loads and executes PalCom components in the form of PRC files, e.g. `mycomponent.prc`. As also mentioned above, the Palcom Component Description files, are used to describe the contents of a given PRC file. The PCS files are typically named, e.g., `mycomponent.pcs` or often just `Component` (with no extension). A PCS file can be thought of as a "compiler script" that specifies, among other things, which classes should be included in a component, and which other components they depend on. One PCS file can only function as a "compiler script" for source files in one language. If components written in another language are required, they have to be compiled into separate PRC files and added as required components in the given PCS file.

The PCS files thus list the provided and required interfaces of the component. Currently only the *class names* are listed; a design were explicit signatures for the various methods in the constituent classes can be (optionally) declared in the PCS files too has been made, but not implemented yet.

4.4.9 Services

As defined in the PalCom Open Architecture deliverable [7] one characteristic of a Service is that it is *self-contained*. `Pal-vm` does not have a Service concept (this is provided by the Service Framework on top of the VM). `Pal-vm` instead provides *processes* which can be used to implement services: Processes have the characteristics of being independent from each other by not allowing object-references to one process in another process.

This characteristic of processes is enforced by the structure of the `System` classes and VM primitives, and currently there is no need for the compilers or VM to do extra checking to ensure this.

The Base libraries also provide the `System>>startProcess` and `System>>startProcessByPath: path` methods, that can be used to launch services dynamically, if written using processes.

Cooperation of services residing either in different processes, in different VMs on a single device, or on different devices will be using the communication protocols defined in The PalCom Open Architecture deliverable [7]

It is of course also possible to program more than one service in a single process, if the "self-contained" characteristic is not interpreted as mandating complete separation. For services that cooperate significantly with other services this may be the easiest approach. The services that are together in one process will, however, not be safe from each other in the sense that:

- a fault in one service could crash another service running in the same process.
- for migration of services between devices it is likely that all the services in one process will have to be migrated together as a unit

5 Toolbox overview

During the PalCom project a range of prototypes have been developed to inform and evaluate the overall architecture and core components. Thus, the total toolbox consists of many different pieces of code on different levels as well as knowledge about how to use this code. We use the word 'toolbox' for the collection of various pieces of code, since they do not span one coherent application/system, but rather a subset of the codes pieces can be selected for performing the task at hand.

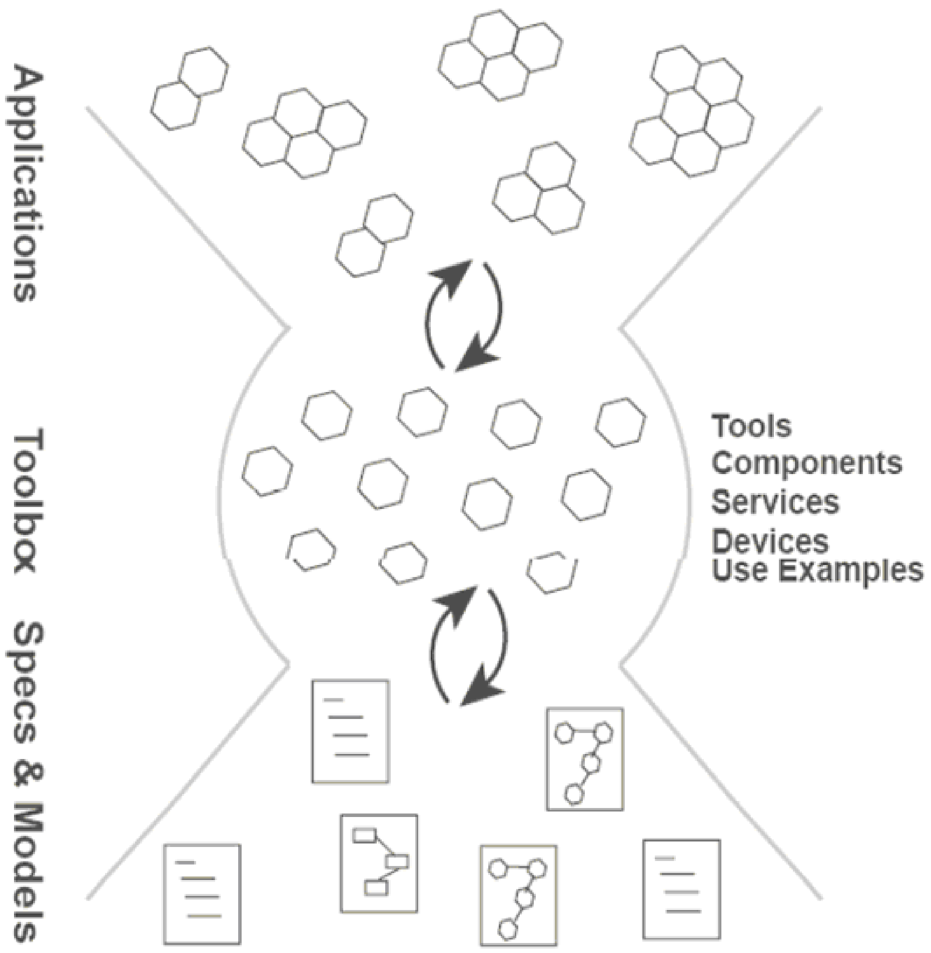


Figure 6: Abstract presentation of the PalCom "Toolbox"

Figure 6 shows a diagrammatic representation of the toolbox. The "pieces of code" (hexagons) in the toolbox have emerged iteratively via (1) implementations of specifications and models from the conceptual frameworks elaborated in the project, and (2) development of the application prototypes (using Participatory Design with end users) incorporating and evaluating the various building blocks from the toolbox.

It is important to keep this image in mind when exploring the toolbox, because every piece of code has usually emerged as a compromise between these two perspectives on the development, and the iterative process is by no means finished.

This section will provide an overview of the toolbox by guiding through the contents of the code base

5.1 Contents of the Code Base File System

The code base for the VM and supporting tools are all in the PalCom repository simply called `palcom`. See section 6 below for instructions on how to check out the repository.

This structure, at top level, has been adopted, resembling the most well-known top level structure for open-source projects:

```
/palcom/
    /bin
    /developer
    /lib
    /doc
```

palcom/bin and palcom/lib ...

...contains scripts and binary files under Subversion, which allows use of a fresh check-out "out-of-the-box" (without installing Eclipse etc.), but still with full access to the sources (unlike traditional binary distributions). You simply add "palcom/bin" to your *PATH* and you will have access to all the "pal-*" scripts, that will execute the executable files appropriate for your platform from subdirectories of "palcom/bin" and "palcom/lib". These will be the latest stable binaries.

palcom/doc ...

contains various documents describing the toolbox and the use of it, including javadoc, text files, tutorials and this Developer's Companion (TeX document).

palcom/developer ...

...contains all sources, projects, resources etc. for the toolbox. An expansion of the folder is shown to the right in figure 7. The contents of the various folders will be described in section 5.2.

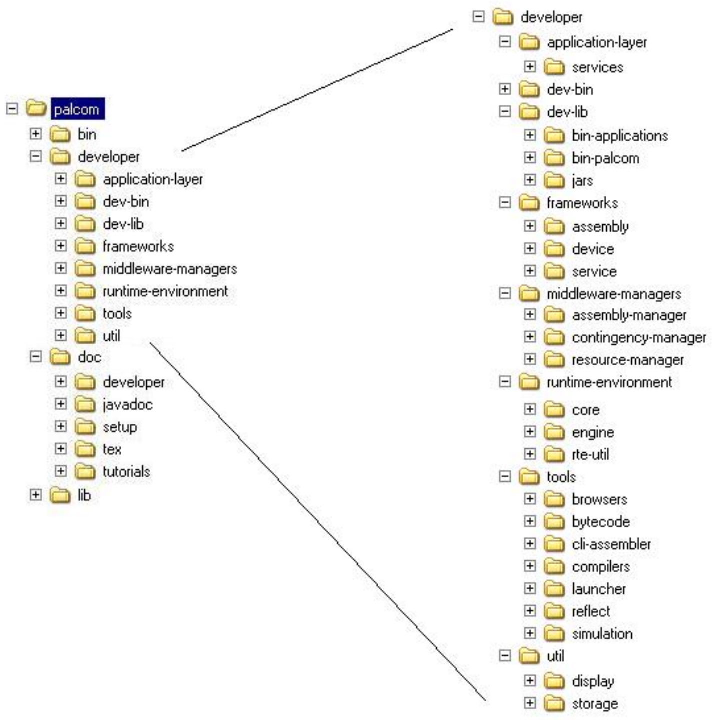


Figure 7: Overview of the palcom repository

palcom/developer/dev-bin . . .

. . . contains a number of scripts, "palcom/developer/dev-bin/pal-dev-*", by use of which one will invoke the latest build in your personal checkout instead of the binary check-out. These scripts requires full installation of Eclipse and CDT plugin (see palcom/doc/setup/Eclipse-install.txt and palcom/doc/setup/Eclipse-SVN.txt) but allows for experiments with and development of the tools, including using Debug builds. Since the developer scripts are differently named than the regular scripts, adding both "palcom/bin" and "palcom/developer/dev-bin" to your PATH should pose no problems.

The scripts are explained in more detail in section 6.5.

5.2 PalCom Developer Code Base

Depending on whether the code is viewed from the file system or via Eclipse, there are two different "topologies": A flat structure if using Eclipse, or a tree structure if using a full repository checkout. Thus, the names of the Eclipse projects all correspond to a folder in the repository.

Figure 8 gives an overview of the different projects.

This section provides a brief description of the separate elements of the developer code base.

5.2.1 palcom/developer/application-layer

The directory contains the various assemblies and services (and simulated devices) provided in the toolbox, i.e. all packages located under `ist.palcom.services` in the javadoc API.

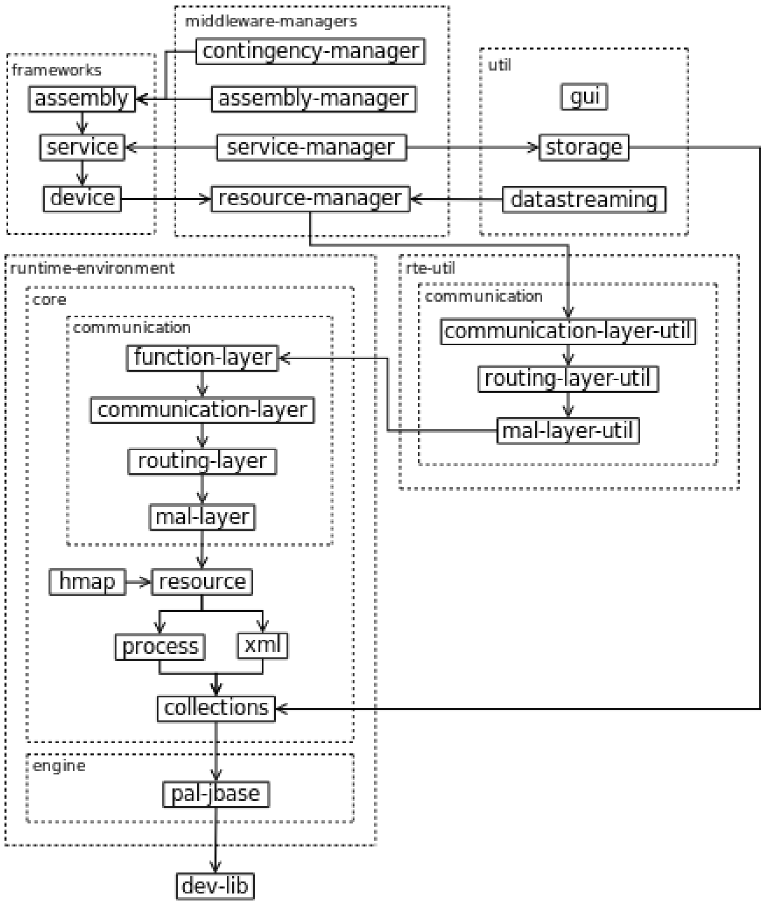


Figure 8: Overview of the projects in the (flat) eclipse structure

5.2.2 palcom/developer/middleware-managers

The directory contains implementation of the various middleware managers, i.e.:

- Assembly-manager
- Resource-manager
- Service-manager
- Contingency-manager

These are described in the PalCom Open Architecture Specification, [7].

5.2.3 palcom/developer/util

This directory contains various utilities to use with the PalCom Architecture.

There is a small GUI component written in Smalltalk for use on small devices. It is decribed in the PalCom Open Architecture Specification, [7], and a tutorial can be found in "palcom/doc/tutorials/gui/nanoX/nanoX-tutorial.txt".

The PalCom storage components are utilities available for arbitrary storage of data for use by PalCom Services and PalCom Assemblies. The storage service has several abstractions and could be used from the applications as well as services and assemblies layers. The Storage component serves both as a basic component in the Palcom framework but has also some specific features to meet the Palcom Challenges. Construction - de-construction is one example where actual device configuration needs to be stored on a device. Another example is Change - Stability where a Palcom application naturally need to handle dynamic as well as persistent data. However, even more important, data also need to move seamlessly between devices and hence need the storage components facilitate such data migration. In the first section we will outline the Basic Palcom Storage Service. The Basic Storage component is layered in several layers of functionality that provides different kinds of abstractions to the core services:

- Palcom File System (PFS) - Base layer for Palcom Storage Service
- Palcom IO (PIO) - I/O Abstactions of Palcom File System
- Palcom Persistent Service (PPS) - Serialization of Palcom Objects
- Palcom Tags (PTS) - Tagging of Persistent Palcom Objects

The next section will breifly describe a scenario of use of the Storage component. This will clearify some aspects of the use of the Storage component API. The last section will describe some work in progress that extend the Storage Component into seamlessly handle the distributed nature of Palcom systems.

The Storage components are also described in more detail in the PalCom Open Architecture Specification, [7].

5.2.4 palcom/developer/runtime-environment

This directory contains three subdirectories:

- "Core", containing various core components for the runtime environment such as the communication and process models and the hgraph structures.
- "Engine", see below.
- "rte-util", see below.

Directory: palcom/developer/runtime-environment/engine

- `palcom/developer/runtime-environment/engine/pal-vm/`
  The `src/VM` subdirectory contains the `pal-vm` (C++) source code. Most files are platform independent. The exceptions are:

GNUMakefile
Contains most of the platform specific settings for all platforms.

Previously Makefiles automatically generated by Eclipse for each platform were used. Although convenient with the user interface for changing platform settings, it turned out to be too time-consuming to maintain settings for each platform individually. Especially since it is not easy to change for other platforms, than the one you are developing on.

To overcome these difficulties, a single GNU Makefile was produced, that basically contains three sections: A section with platform specific settings of compile time options, includes etc., a section with a common listing of the constituent source files, and a section with make targets for the user to invoke.

This simple structure has made it much easier to add new platforms, or to change settings for all or specific platforms.

Platform.hpp
Definition of interface to platform specific operations used by the VM. This includes some timing functionality as well as basic file I/O.

UnixPlatform.cpp
UNIX implementation of platform specific functions. Used on Linux and Mac OS X platforms.

CygwinPlatform.cpp
Cygnus Windows implementation of platform specific functions. Since Cygwin is an attempt to mimic UNIX on Windows, the differences from UnixPlatform.cpp are very small, and mostly concern various constants and path notation.

NativeSymLookup.cpp
This is used for looking up symbols after loading third party libraries. This is needed when a PalCom component is loaded, that references external third party code using the Native interface. This file should be split in parts corresponding to each platform, and included in the above mentioned platform specific source files.

Threads.cpp
The implementation of threads currently assumes POSIX Threads to be available. This may not always be the case, and as such this may need to be split out into the platform specific files.

Besides the above mentioned files, a few files use non-ANSI C++ code. Specifically the GCC [3] extensions for *computed goto*, *address-of labels*, and *asm bindings of variables to hardware registers* are used. All of these usages are, however, only for optimisation purposes, and ANSI C++ compliant code is always included as fallback these places in the code. So even for a non-GCC compiler, this code should work as-is.

- `palcom/developer/runtime-environment/engine/pal-vm/benchmarks`
  Smalltalk programs used to measure the performance of the VM and compilers.

- `palcom/developer/runtime-environment/engine/pal-vm/tst`
  Smalltalk programs used as a unit test for testing the functionality of the VM and the `pal-st` compiler.

- `palcom/developer/runtime-environment/engine/j-libs/`
  Basic libraries - written in a mixture of Smalltalk and Java source code - to be used on top of the `pal-vm`. Contains a small subset of the standard Java class libraries. Mainly used to allow for execution of Java programs and libraries on `pal-vm`, but can also be used from Smalltalk.

- `palcom/developer/runtime-environment/engine/jbenchmarks/`
  Java programs equivalent to the Smalltalk programs in `palcom/developer/runtime-environment/engine/pal-vm` `benchmarks` used to compare the performance of programs written in Java source code, with equivalent programs written in Smalltalk

- `palcom/developer/runtime-environment/engine/jtst/`
  Java programs used as a unit test for testing the functionality of the VM and the `pal-j` compiler.
- `palcom/developer/runtime-environment/engine/pal-base/`
  Minimal base libraries used by programs running on `pal-vm`. Contains base classes like `Object`, `Integer`, `Boolean`, etc., but also slightly more complex classes like `CheckedArray`, `Process` and the `System` class.
  Furthermore the subdirectories `palcom/developer/runtime-environment/engine/pal-base/networking/` and
  `palcom/developer/runtime-environment/engine/pal-base/storage/` contains classes supporting networking and (very simple) disk-based storage.
- `palcom/developer/runtime-environment/engine/pal-jbase/`
  Java library corresponding to `pal-base` to be used when running a (Java) PalCom program on the JVM platform.
- `palcom/developer/runtime-environment/engine/thirdparty/`

Directory: palcom/developer/runtime-environment/rte-util

5.2.5 palcom/developer/tools

The directory contains various PalCom tools to make usage of the PalCom infrastructure easy and efficient. Examples are the different assembly browsers, compilers for different programming languages and the launcher application making it easy to deploy services and assemblies.

Furthermore the tools include some byte-code tools for handling the `pal-vm` bytecodes, etc.:

- `palcom/developer/tools/bytecode/pal-asm/`
  The (Java) source code of the bytecode assembler.
- `palcom/developer/tools/bytecode/pal-dis/`
  The (Java) source code of the bytecode disassembler.
- `palcom/developer/tools/bytecode/pal-bytecodes/`
  Common bytecode (Java) libraries used by the bytecode assembler, disassembler, and the Smalltalk compiler.
- `palcom/developer/tools/reflect/pal-dep/`
  The (Java) source code of the PalCom component dependency analyzer
- `palcom/developer/tools/compilers/pal-j/`
  The (Java) source code for the PalCom Java subset (PalJ) compiler. It contains two main subdirectories: One for a generic Java 1.4 frontend, and one for a backend producing `pal-vm` bytecodes. Both of these are written using the JastAdd [10] Java based attribute grammar system. The Java frontend is a copy from another project, whereas the backend is developed in PalCom.
- `palcom/developer/tools/compilers/pal-st/`
  The (Java) source code of the PalCom Smalltalk compiler. The frontend of this compiler dates back to an older project called SOM [9], and its successor called POMP [5][8], whereas the PalCom bytecode backend is developed in PalCom.

5.2.6 palcom/developer/frameworks

The PalCom frameworks for devices, services and assemblies have been described in the programming models in section 4.

6 Installation guides

The PalCom open source code base is available for checkout using Subversion (`http://subversion.tigris.org`). The subsequent sections describe how to get, install and build the system.

6.1 Checking out the code

Instructions containing a very brief introduction to Subversion, how to get subversion for your machine, and descriptions on how to check out the palcom code repository can be found on

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Palcom-SVN.txt`

For convenience the text is repeated below. The instructions for checking out the code included here is based on command-line checkout. Notice, that on Windows, you will have to install Cygwin to do this command-line checkout – see section 6.6.

For checkout using Eclipse, refer to section 6.3 below, but if you are new to Eclipse, you may want to refer to section 6.2, also below, first.

```
File: http://svn.ist-palcom.org/svn/palcom/doc/setup/Palcom-SVN.txt

Introduction to using the PalCom Subversion (svn) Repository.

1. Subversion Introduction

   Subversion (a.k.a. SVN) is meant as a replacement for CVS.
   It has almost the same commands as CVS, but is improved in a number
   of ways, notably directory handling, binary files handling and
   offline mode.
   Here are some good references:

   - Subversion Home
     http://subversion.tigris.org/

   - Version Control with Subversion
     http://svnbook.red-bean.com/en/1.0/index.html
     Free online "Bible"

   - Subversion for CVS Users
     http://svnbook.red-bean.com/en/1.0/apa.html
     Appendix of above book, with important differences explained

   - Subversion for CVS Users
     http://osdir.com/Article203.phtml
     Another short guide for previous CVS users

   - Subversion Quick Reference Card
     http://www.cs.put.poznan.pl/csobaniec/Papers/svn-refcard.pdf
     2 page summary of commands


2. The PalCom Subversion Repository Location

   The repository is located in

     http://svn.ist-palcom.org/svn/palcom


3. SVN usernames

   In order to be able to commit changes to the repository, you must
   be a registered user.

   Contact <palcom2-admin@ist-palcom.org> to get a username.

   For read-only checkout of the trunk, you need not a username.

4. Command line checkout

   If you have installed the command line tool "svn" (see Tools
   below), you can check out the main trunk of the repository read
   only into a local directory named "palcom" as such:

      svn checkout http://svn.ist-palcom.org/svn/palcom/trunk palcom

   Only the trunk can be checked out anonymously.
   If you are a registered developer, you can check out with
   permission to commit as such:

      svn checkout --username <USERNAME> http://svn.ist-palcom.org/svn/palcom/trunk palcom

   Checking out into a local directory named "palcom" is currently
   required for the build process to work.

   For non-registered developers, the username "anon" with password
   "anon" can also be used. This gives read-only rights, just like
   true anonymous checkout. The main use of this is from the Eclipse
   Subclipse plugin, that always asks for a username on palcom
   checkout (see Eclipse-SVN.txt).

5. WWW Access to the Repository

   The repository can be browsed at

      http://svn.ist-palcom.org/svn/palcom

6. Using SVN with Eclipse.

   See http://svn.ist-palcom.org/svn/palcom/doc/setup/Eclipse-SVN.txt

7. Tools
```

```
 - Platform independent:

   Eclipse:
   - Plugin:
     http://subclipse.tigris.org/
     or, alternatively use:
     http://www.polarion.org/index.php?page=overview&project=subversive


   RapidSVN:
   - http://rapidsvn.tigris.org/

   SmartSVN:
   - http://www.syntevo.com/smartsvn/index.jsp

 - Windows:
   - Explorer extension:
     http://tortoisesvn.tigris.org/
   - Using Subversion On Windows:
     http://weblogs.asp.net/nleghari/articles/subversion.aspx
   - cygwin:
     http://www.cygwin.com/setup.exe, package "subversion"

 - Macintosh:
   - Finder extension:
     http://scplugin.tigris.org/
     http://www.eyefodder.com/blog/2007/06/subversion_and_finder_integrat.shtml
   - Subversion With Mac OS X Tutorial
     http://www.rubyrobot.org/tutorial/subversion-with-mac-os-x
   - Subversion on Mac OSX:
     http://forevergeek.com/geek_resources/subversion_on_mac_osx.php
   - How to Install Subversion on Mac OS X
     http://www.wikihow.com/Install-Subversion-on-Mac-OS-X
   - svnX:
     http://www.lachoseinteractive.net/en/community/subversion/svnx
   - Xcode:
     http://developer.apple.com/tools/subversionxcode.html

 - Linux:
   - Nautilus GNOME extension:
     http://naughtysvn.tigris.org/
   - A subversion client for KDE
     http://kdesvn.alwins-world.de/

8. Content of Repository
   The following briefly describes the content of the Repository.

   8.1 Overview
       The basic structure is simple:

          palcom/
             bin/
             doc/
             lib/
             developer/
               application-layer/
               middleware-managers/
               frameworks/
          util/
          runtime-environment/
            engine/
            core/
            rte-util/
          tools/
          dev-bin/
          dev-lib/

    See palcom/doc/setup/figures/projects.png for a graphical
    overview, and see PalCom Open Source Doc (Working Note 117)
    for a detailed walk-through of the Repository.

  8.2 Explanation
    A traditional top level structure has been adopted,
    resembling the most well-known top level structure
    for open-source projects.

    palcom/bin and palcom/lib:
       contains scripts and binary files, which allows
       use of a fresh check-out "out-of-the-box" (without
       installing Eclipse etc.), but still with full access to the
       sources (unlike traditional binary distributions).
       You simply add "palcom/bin" to your PATH and you will have
       access to all the "pal-*" scripts, that will execute the
       executable files appropriate for your platform from
       subdirectories of "palcom/bin" and "palcom/lib".
       These will be the latest stable binaries.

    Developer tools: palcom/developer/dev-bin
        By using the "palcom/developer/dev-bin/pal-dev-*" scripts, one
        will instead invoke the latest build in your personal
        checkout.
        Requires full installation of Eclipse and CDT
        plugin (see palcom/doc/setupt/Eclipse-install.txt and
        palcom/doc/setup/Eclipse-SVN.txt) but allows for
        experiments with and development of the tools, including
        using Debug builds.
        Since the developer scripts are differently named than the
        regular scripts, adding both "palcom/bin" and
        "palcom/developer/dev-bin" to your PATH should pose no problems.
    palcom/developer
        Contains all sources, projects, resources etc. for the system
```

6.2 Installing Eclipse

Much of the code in the PalCom open source repository is written in in Java or C++. The code has been organized for convenient work using the Eclipse development environment, see `http://eclipse.org`. Instructions for how to install Eclipse and required plugins are located at

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Eclipse-install.txt`

For convenience the text is repeated below.

```
File: palcom/doc/setup/Eclipse-install.txt

1. Getting Eclipse

   A number of the tools for PalCom are developed using the Eclipse IDE.
   The PalCom projects are known to work with Eclipse version 3.3.

   To get Eclipse, go to

      http://www.eclipse.org/downloads/

   and follow the instructions.

2. Getting the C++ plugin (CDT) for Eclipse

   For the PAL-VM virtual machine, you will furthermore need the C++
   Developer Toolkit (CDT) plugin.

   The procedure is described at, e.g.
      http://www.mirrorservice.org/sites/download.eclipse.org/eclipseMirror/technology/phoenix/demos/install-cd
   (Flash Player browser plugin required)

   The essentials are:

   Inside Eclipse do this

   a. Select Help->Software updates->Find and install
      (meaning: In menu "Help", select item "Software updates", and select
      submenu item "Find and install")

   b. Choose "Search for new features to install"

   c. Select "Callisto Discovery Site", press Finish

   d. Select a Callisto mirror site (e.g. the default)

   e. Unfold the Callisto Discovery Site tree presented, unfold the C
      and C++ Development leaf, and choose the newest version of CDT and
      press Finish.

3. Using SVN in Eclipse

   See Eclipse-SVN.txt

4. Java compiler version

   There has been made an agreement to allow using Java 5.0 for the
   Java source files in PalCom (generics, annotations, etc.).
   And for some tools, this is currently being used.
   In eclipse this means that one should go to

     Window->Preferences->Java->Installed JREs
         and select JRE 1.5 as default and
     Window->Preferences->Java->Compiler
         and set Compiler Compliance Level to 5.0

   Furthermore you should add the appropriate Java 5.0 paths to your
   PATH environment variable.
   Please notice, that pal-j currently does NOT support Java 5.0
   features, so you cannot use these in source code that is to be
   compiled for running on the pal-vm itself.
   An augmented pal-j compiler with 5.0 features is expected started
   on mid 2006.
```

By design Eclipse works with *projects*. The palcom open source repository has been divided into a number of such projects, allowing developers to use only a subset of these. However, this you get a flat directory structure in your Eclipse workspace, unlike the nested structure shown in Figure 7.

6.3 Getting the code into Eclipse

You may use Eclipse to do the SVN checkout of the Palcom Repository instead of having to use command line commands. How to do this is explained in

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Eclipse-SVN.txt`

which for convenience is repeated below.

Instead of individually checking out each Eclipse project separately, we the procedure described includes checking out the whole Repository as one "project" and then importing all contained project into Eclipse.

```
File: http://svn.ist-palcom.org/svn/palcom/doc/setup/Eclipse-SVN.txt

1. Eclipse SVN Plugin
---------------------

To use SVN from within Eclipse you have to manually install a plugin
for this.

The suggested plugin is found at

    http://subclipse.tigris.org/

To install it, inside Eclipse do this:

   a. Select Help->Software updates->Find and install
      (meaning: In menu "Help", select item "Software updates", and select
      submenu item "Find and install")

   b. Choose "Search for new features to install"

   c. Press "New Remote Site"

   d. Enter URL http://subclipse.tigris.org/update_1.2.x, press OK

   e. Make sure "Subclipse" is checked. Press "Finish"

   f. When "Select the features to install" appears, click the + and
      select Subclipse. Press "Next".
   g. Accept the license and press "Finish". Eclipse will suggest
      restarting.


2. Eclipse SVN and Command line svn
-----------------------------------

Unlike the CVS plugin, there should not be any problems switching
between Eclipse and command line, since both use the HTTP protocol to
access the repository.

3. Checking out "palcom" and importing Eclipse Projects.
--------------------------------------------------------

The SVN file structure of the PalCom toolbox does not follow the
(flat) directory structure preferred by Eclipse projects.
For this reason, you need to perform a few extra steps to establish
the Eclipse projects contained within the SVN Repository.

We here describe some alternative ways to do this.

2.1.  Checking out "palcom" and importing

There are (at least) two ways to check out the full source tree.
These alternatives are described in 3.1.1 and 3.1.2.  After having
done one of these, you continue with the import described in 3.1.3,
after which you should check your settings as described in
Eclipse-build.txt.

3.1.1 Checking out "palcom" with command line

      You can start by checking out the full main trunk of "palcom"
      using command line svn as described in Palcom-SVN.txt.
      Then go directly to 3.1.3 below.

3.1.2 Checking out "palcom" from within Eclipse

      If you do not have (or do not wish to use) a command line
      svn, you can use Eclipse to check out "palcom".
      However, since Eclipse insists on everything being organized
      as projects, and due to a bug in Eclipse (on some platforms)
      this is somewhat cumbersome. Here we go:

      A. First change your workspace to something else:

         File->Switch Workspace...

         If you do not have an existing workspace you do want
         to "pollute", simply use the Browse button to select a new
         location for this (somewhat temporary) workspace.
         [This step is necessary due to a bug in Eclipse (3.1.2) on
         at least Windows and linux: Eclipse will complain about
         "Overlapping workspaces" if you try to import projects as
         described in 3.1.3 in the same workspace]

      B. Set up the SVN Repository location.

         Change Perspective to the SVN Repository Exploring
```

```
 perspective (normally a button/menu in the upper right corner
 of the Eclipse window, showing "Java", "Resource" or some
 other current Perspective).
         Right click in the empty part the left pane "SVN Repository",
 and select
           New->Repository Location...
         Specify the URL
           http://svn.ist-palcom.org/svn/palcom/trunk
         The plugin will ask for your SVN username and password. If
         you are a registered PalCom developer, use your assigned
         username and password. Otherwise, enter username "anon" and
         password "anon" for read-only access (without commit rights).

      C. Now right-click on this new location in the left pane and choose
         "Checkout" from the menu. Name the project "palcom".
         If you switch to the Resource view, you will see a
         "project" called "palcom" in this temporary workspace.

      D. Finally switch your workspace back to the workspace you
         want to import the Eclipse project into, and go to section
         1.1.3, here you should of course the use the other
         "temporary" workspace location to import from.

      E. After importing the projects, you can actually delete the
         checkout in the "temporary" workspace (but choose NOT to
         delete contents on disk!), but we recommend keeping it for
         easy update of the full source tree from within Eclipse.

      Now continue with 3.1.3.

2.1.3 Importing Eclipse projects from a full "palcom" checkout

      You should first have checked out the full SVN tree using one
      of the methods described in 3.1.1 or 3.1.2.

      In Eclipse open

      File->Import...->Existing Projects Into Workspace

      Using the Browse button you navigate to the "developer"
      directory in your "palcom" checkout. Then make sure the
      correct projects are found and checked on the list presented,
      and press Finish. It is safe to import all the listed
      projects. After the import, automatic builders will be
      triggered for some projects. When they have finished, some
      errors will remain. Please see Eclipse-build.txt for
      information about how to perform the final build step
      manually.
```

6.4 Building from Eclipse

Once you have checked out the Palcom Repository and imported the projects into Eclipse, as described above, you are ready to build and execute the various projects. Due to a complicated set of project dependencies, Eclipse is not able to automatically build all the projects in the right order as they are imported. Thus multiple build errors will be indicated in your workspace after the import has finished. The solution is to run two ant scripts, as it is decribed in

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Eclipse-build.txt`

which for convenience is repeated below.

```
File: palcom/doc/setup/Eclipse-build.txt

Overview of how to build the PalCom projects in Eclipse

1. One-step build of the projects from inside Eclipse

   Please see Eclipse-SVN.txt for information about how to check out
   and import projects into your Eclipse workspace. Project builders
   will be triggered automatically after the import, but some errors
   will remain when they have finished. It may be most convenient for
   you to disable this automatic build (remove the check mark in the
   "Project" menu).

   In order to get rid of the errors, you need to perform one manual
   step: re-building the "dev-lib" project. Right-click on
   dev-lib/build.xml, and choose Run As->Ant build... (three dots).
   On the Refresh tab, check "Refresh resources upon
   completion" and "The entire workspace". Click Apply and then
   Run. After the dev-lib Ant Build has finished, there should
   be no errors. Enable automatic build again.

   Note: cases have been observed where the initial run of the dev-lib
   Ant script fails with an "out of heap space" error during a
   compilation step. If that occurs, re-run the Ant build..., and this
   time put "-Xmx500M" in the "VM arguments" field on the JRE tab
   before clicking Run. This heap space setting will be saved for
   later runs.

   Section 3 describes in more detail how the different projects are
   built.

2. Building for JVM on the command-line

   If you want to work from the command line, the projects can be
   build for execution on the JVM using

       pal-dev-build -J

   (see Command-line-tools.txt for instructions on how to set up for
   command line execution).
   If you do this, the next time you start up Eclipse, make sure to do
   a Refresh of all projects, to make Eclipse aware of the build you
   did.

3. Building the projects

   3.1 pal-vm
```

Notice that if you import the "pal-vm" project, you should also make sure to have the CDT plugin installed. See Eclipse-install.txt.

3.2 AST classes for the pal-j compiler

If you import the pal-j compiler project Backend, its Ant script build.xml has to be run for generating the compiler's AST classes (defined in JastAdd grammars and aspects). This is done automatically after import. You can also choose Project->Clean with Backend selected, or right-click on build.xml and select Run As->Ant Build.

When running the Ant script manually, it is recommended to set up an Eclipse launch configuration like this: right-click on Backend/build.xml, and select Run As->Ant Build... (three dots). On the Refresh tab, check "Refresh resources upon completion" and "The entire workspace". Click Apply and then Run or Close.

3.3 Pal-J projects

Pal-J projects are built for the JVM using a combination of two tools: the Eclipse compiler and Ant (Pal-VM builds are run from the command-line). The Eclipse compiler runs automatically when a source file is changed in Eclipse (provided that Project->Build Automatically is checked). The generated class files end up in the bin directory in each project (which is hidden in the Package Explorer view, but can be seen in the Navigator).

Ant scripts, named build.xml, are placed in the project directories of some projects. For projects that have JastAdd classes, their Ant scripts build those. The Ant script in the dev-lib project (dev-lib/build.xml) compiles code for all the projects into two JAR files. The JAR files are used by the EclipseBrowser, and also when running from the command-line. This Ant script can be run by selecting Run As->Ant Build, as described for Backend in 3.2.

If you have the Ant tool installed, the Ant scripts can also be run from the command-line, by typing "ant" when standing in the dev-lib project directory. See http://ant.apache.org/ for information about how to install and use Ant.

The dependencies between the Pal-J projects for the PalCom Runtime Environment, for Middleware Managers, for Frameworks and for Utilities are illustrated in the figure palcom/doc/setup/figures/projects.png. The projects are the boxes with filled edges. A project can use classes in all projects directly or indirectly referenced in the graph (following the arrows one or more steps). These dependencies are captured in the Java Build Path settings in each project (and thus in the Eclipse .classpath files), in the following way: all projects are set to export the projects they depend on, and each .classpath file only declares the project's direct dependencies in the graph. The projects in the engine box without illustrated dependencies contain the pal-vm source code and base libraries.

The pal-jbase project is a JVM implementation of (a subset of) the Smalltalk classes in pal-base. It uses the whole JDK, because the classes in pal-jbase only run on the JVM. From collections and up, the classes should run on both the Pal-VM and the JVM. The collections project uses a subset of the JDK classes in java.lang, java.io and java.util. The rest of the JDK classes are excluded from use in Eclipse (an exclusion filter in the file collections/.classpath). For projects above collections, you get Access restriction errors in Eclipse if you try to use classes from the JDK, except the selected classes in java.lang, java.io, and java.util.

For the Ant scripts, the classpath dependencies are not managed on a project-by-project basis. Instead, the combination of all source folders and all classpaths are listed in one place, in the dev-lib Ant script (dev-lib/build.xml). The path elements "palcom.src.path", "application.src.path" and "class.path" define the source paths and classpaths.

2.4 AST classes for Pal-J projects

The projects mal-layer, communication-layer, function-layer and assembly have JastAdd classes. If you import these projects, you can re-generate the JastAdd classes by running Ant scripts. Re-generation is necessary if you edit JastAdd source files (.jadd, .jrag or .ast files). A quick way to re-generate all JastAdd classes is to run the jastadd-gen target in dev-lib/build.xml.

2.5 The EclipseBrowser plugin

If you import the EclipseBrowser plugin, you need the jars-plugin project, which exports a generated JAR file used by the EclipseBrowser. The JAR file can be re-built by running the Ant script EclipseBrowser/build.xml. It is recommended to set up a launch configuration with "The entire workspace" chosen on the Refresh tab, as described above. Rebuilding the JAR file is necessary if you make changes to any of the Pal-J projects, discussed under 2.3, while working with the EclipseBrowser.

Cases have been observed where the jars-plugin project does not find the JAR file in Eclipse, even after it has been generated, and after refreshing the project (a "missing required library" error). A workaround in such a case can be to make a small change to jars-plugin/build.properties (just add a space somewhere) and save the file. That makes Eclipse see the JAR file again.

For further information about how to set up and use the EclipseBrowser, see the documentation files in EclipseBrowser/doc.

6.5 Using command line scripts

To ease the execution of the various tools in the code base, as well as the building of the code base, a number of command line scripts are available.
The use of these is described in

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Command-line-tools.txt`

which for convenience is repeated below:

```
File: palcom/doc/setup/Command-line-tools.txt

--------------------------------------------------------------------------------
Introduction
--------------------------------------------------------------------------------

The PalCom developer tools can be invoked from the command line on
Windows, Linux and Macintosh (OSX).
These command line tools require a BASH shell to be executed.

On Linux and Macintosh, these are normally available with the standard
installation.

On Macintosh you run bash through the Terminal.

On Windows the tools depend on the Cygwin UNIX emulation to work.
See Cygwin-Install.txt for details on how to install and configure
this.
Some scripts can also be invoked in the standard Windows command
interpreter CMD.exe using BAT files.

--------------------------------------------------------------------------------
A. User tools
--------------------------------------------------------------------------------

The scripts placed in "palcom/bin" use precompiled, binary distributed
tools, and can thus can be used "out-of-the-box" for a freshly checked
out palcom directory.

They are:

  pal-vm[.bat]
    Invoke the binary distributed Palcom Virtual Machine.
    Script command line options allows selecting between four
    different builds of the VM (Release, Debug, Checked, Profiling).
    Try "pal-vm -h"
  pal-j[.bat]
    Invoke the binary distributed Palcom Java Compiler.
    Try "pal-j -help" or see palcom/doc/tutorials/pal-j
  pal-st[.bat]
    Invoke the binary distributed Palcom Smalltalk Compiler.
    Try "pal-st -h" or see palcom/doc/tutorials/pal-st
  pal-asm[.bat]
    Invoke the binary distributed Palcom Bytecode Assembler.
    Try "pal-asm -h"
  pal-dis[.bat]
    Invoke the binary distributed Palcom Bytecode Disassembler.
    Try "pal-dis -h"
  pal-dep
    Invoke the binary distributed Palcom Component Dependency analyzer.
    Try "pal-dep -h"
  pal-ovb
    Invoke the binary distributed Palcom Overview Browser.
  pal-ass
    Invoke the binary distributed Palcom Assembly Assembler.
    Type "help" at the "> " prompt to get a list of commands it handles.
  pal-launcher
    Invoke the binary distributed Palcom Assembly and Service Launcher.

To use these scripts, simply add "palcom/bin" to your search PATH.

Here is a Cygwin alias that does it (assuming Palcom checked out in C:\):

   alias pal-setup='export PATH="/cygdrive/c/palcom/bin:$PATH"'

You can also set PATH through the standard Windows interface

   Start->Control Panel->System->Advanced->Environment Variables

A corresponding Linux or Macintosh alias, assuming palcom checked out
in your home directory:

   alias pal-setup='export PATH="~/palcom/bin:$PATH"'

To make a simple test of whether you have set up things correctly, try
invoking

   pal-root

which should simply print out the top directory of you palcom
checkout.

--------------------------------------------------------------------------------
B. Developer tools
--------------------------------------------------------------------------------

The scripts in palcom/bin/developer use the latest build of the tools
in palcom/developer, and as such requires pal-dev-build to be run
before being used. All developer scripts have names that
start with pal-dev-, and it is thus possible to have both the
developer-, and non-developer scripts in ones PATH.

All scripts exists in BASH (for Linux/MacOSX/Windows Cygwin) syntax,
and a few of them also in CMD (.bat files for Windows) format.

These scripts include developer versions of all the "out-of-the-box"
scripts described above, as well as a number of other scripts:

  pal-dev-build
    Builds the basic code base including compilers, VM, base
    libraries, and core libraries. Use a number of other build scripts
    (not listed here) to build the various parts of the code base.
  pal-dev-build-javadoc
```

```
    Rebuilds the javadoc in palcom/doc/javadoc from the current source
    files.
  pal-dev-clean
    Cleans the developer tree of all generated files
  pal-dev-test
    Executes various tests to ensure a consistent code base. This
    includes the tst and jtst unit tests, a number of test programs
    for the core libraries, all the tutorials in palcom/doc/tutorials,
    and (optionally) the benchmarks programs listed above.
  pal-dev-export
    Makes an "export" of the current binaries in the developer tree
    into palcom/bin and palcom/lib.

To use these scripts, add palcom/developer/dev-bin to your PATH as
described for the out-of-the-box scripts above.

Correspondingly, you can do a simple test of the setup by trying

   pal-dev-root

which should simply print out the top directory of you palcom
checkout.

If this works, you should be ready to invoke

   pal-dev-build

which builds all the basic elements of the palcom toolbox for you.
```

As said, the scripts are intended to make life easier for the programmers and users. They act as wrappers for the binaries, selecting the appropriate binaries for the current platform (in the C++ written VM case), setting up appropriate CLASSPATHs for Java programs (e.g. the compilers) and appropriate PalCom component paths, abstract away differences between platforms, like path notations on UNIX versus Windows, and setting default command line options.

6.6 Installing Java/GCC/Cygwin

For working with the PalCom code in Eclipse, you need only install Eclipse itselv, see section 6.2, since Eclipse includes a Java compiler and runtime environment. However, if you checked out all the PalCom projects, as described in section 6.3, you will also have checked out the `pal-vm` project. This requires the C++ CDT extension described in section 6.2 to be installed too, in addition to a working GNU C Compiler (GCC) installation.

If you do not need to run the PalCom Virtual Machine `pal-vm` (see section 4.4.1), you may simply delete this project from you Eclipse project list.

Alternatively you may need to install GCC yourself.

You can get both source code and binary installers for GCC from

- `http://gcc.gnu.org`

For MacOSX GCC is part of the Xcode developer tools. MacOSX does not install the gcc compiler by default but it is freely available in the xcode suite of development tools. To install the gcc compiler, download the xcode package from `http://connect.apple.com`. You will need to register for an Apple Developer Connection account.

Once you've registered, login and click Download Software and then Developer Tools. Find the Download link next to Xcode Tools (version) - CD Image and click it. Find the downloaded package, doubleclick it and follow the installation instructions to install gcc and a host of other development applications. GCC will be located at `/usr/bin/gcc`.

For command line execution on any platform, you also need a separately installed Java Development Kit (JDK) – at least Java SE 1.5. If you have Eclipse installed, it is possible to reuse the Java installation included in Eclipse, but this is not documented here. Instead we refer to the standard place to find it, namely

- `http://java.sun.com/javase/downloads/index.jsp`

For Windows, as noted in section 6.1, command line execution currently relies on the Cygwin environment being installed. Cygwin includes Java and GCC. Instructions for installing Cygwin, as well as some hints on increasing productivity when working within Cygwin can be found at

- `http://svn.ist-palcom.org/svn/palcom/trunk/doc/setup/Cygwin-Install.txt`

For convenience the text is repeated below.

```
File: palcom/doc/setup/Cygwin-Install.txt

-------------------------------------------------------------------------------
Introduction
-------------------------------------------------------------------------------

The PalCom developer tools can be invoked from the command line on
Windows, using the Cygwin UNIX emulation suite.

To install this, download an execute the installer:

http://www.cygwin.com/setup.exe

When run, you will be asked a number of questions.
Typical answers are:

  Download Source: Install from Internet
  Root Directory: C:\cygwin
  Install for: All Users
  Default text file type: DOS
   (IMPORTANT if you check out palcom using a windows tool,
    e.g. if you use Eclipse as described in Eclipse-SVN.txt)
  Local Package Directory: <anywhere on your computer>
  Internet Connection: Direct Connection
  Download site: <a site near you>

When you reach the Select Packages list, there are a number of
packages you need to install, as well as some we recommend you also
install. You may wish to click the "View" button one or more times
until an alphabetical list of packages is shown.
To include a package in the installation, click the two "circular
arrows" once to get the latest stable release instead of "Skip".

The packages besides the default ones selected by the installer
itself we recommend are:
  + subversion     (if you want to use cygwin for svn checkout/update/commit)
  + gcc-g++    (needed to compile pal-vm)
  + less    (file paging tool)
  + make           (needed to compile pal-vm and some tutorials)
  + perl           (needed for a few scripts)
  + vim/emacs/nano (text editors)

Once you press Next, the relevant packages will be downloaded and
installed.

Installing ANT
--------------

You will need ant to build the java based projects in palcom.
Ant seems not to be included in the current cygwin packages, but can
be downloaded from

http://ant.apache.org/bindownload.cgi

Setting up for ant is rather simple, see

http://ant.apache.org/manual/index.html

For cygwin it basically amounts to (assuming ant installed in C:\)

export ANT_HOME=/cygdrive/c/ant
export PATH=${PATH}:${ANT_HOME}/bin

Installation check
------------------

Please do this in a cygwin bash window:

which make

This should report

/usr/bin/make

Using the cygwin built-in 'make' is essential to a correct build of
the pal-vm and some of the tutorials.
If you get another path from "which make", please fix your PATH (see
below for examples).

Cygwin tips
-----------

Quickedit:

If you click the cygwin icon in the upper left corner of a cygwin
window, you get a menu ending in "Properties".
Select Properties, and under "Edit Modes" select Quickedit.
When you press OK select "Modify icon that started this window".
Then the current and future cygwin windows will support copy and paste
using the mouse: Drag to select, right-click to copy and right-click
to paste.
```

```
Home directory
--------------

If you add a Windows Environment Variable called "HOME" via
Start->Control Panel-System->Advanced->Environment Variables
you can specify where cygwin windows start up, and where cygwin
"dotfiles" reside.
We recommend HOME=C:\

Dotfiles: .bash_login
---------------------

The most important dotfile you can place in HOME is called
.bash_login.
In here you can e.g. specify variables to set during bash shell
execution. Some useful examples:

# Include SUN Java installation in setup
# Notice the alternative cygwin notation for C:\
export JAVAHOME="C:/Program Files/Java/jdk1.5.0_11";
export PATH="/cygdrive/c/Program Files/Java/jdk1.5.0_11/bin:$PATH";

# Include palcom in setup
export PATH="/cygdrive/c/PalCom/palcom/bin:/cygdrive/c/PalCom/palcom/developer/dev-bin:$PATH";


Dotfiles: .inputrc
------------------

In HOME you may also place a file called .inputrc, which modifies the
way the command interpreter in cygwin bash works.
Here is an example content:

set completion-ignore-case On
set show-all-if-ambiguous on
"\e[A": history-search-backward
"\e[B": history-search-forward

The first line augments the completion functionality to be case
insensitive.
That is, if you have a file like "Cygwin-Setup.txt" in your current
directory, and on the command line type

ls cyg<TAB>

this will complete to

ls Cygwin-Setup.txt

The second line specifies that you want all possibilities listed, if
there are more than one matching completion.

The two next lines specify that arrow-up and arow-down can be used to
complete via the command history. That is, if you previously did a
command like

ls Cygwin-Setup.txt
```

```
and then later type

ls<arrow-up>

this will complete to

ls Cygwin-Setup.txt

(assuming that was the latest commend starting with "ls").

For more tricks like this, refer to "man bash".
```

---

7 'Tools' in toolbox

7.1 PalCom Overview Browser: User's guide

The PalCom Overview Browser has been described in D49(ref), the text is repeated here for convenience.

Complementing the developer-centric perspective put forth by the Palcom Developer's Browser, the Palcom Overview Browser (OVB) focuses on bringing a limited subset of the Developer's Browser to the non-technical user. The OVB thus currently allows users to inspect the topology and some internal state of Palcom devices visible on the network. OVB is an example of use of the *Pal-Visualization* Remote Visualization Framework described in section 9.1.1 of Deliverable 54

As such, the OVB is a Palcom device (OVBDevice) with an assembly manager maintaining an OVB assembly (named VizBrowser3) currently consisting of a Swing Display Service and a Visual Browser Accomplice Service, both also running on the device. The Visual Browser Accomplice Service listens for discovery events about devices, services, assemblies and connections and creates an UI based on this information on the canvas supplied by the Swing Display Service. As such, the point of origin is the displaying of discoverable devices and along with and between them, any service, assembly and connection.

Once started, the OVB will appear as a window with a dark green background and a series of minimized windows indicative of the devices discovered. As new devices are discovered, they will also appear as windows. These windows can be dragged around and resized as the user sees fit. Lines between devices indicate connections between services and assemblies running on them.

Once expanded, the device windows will yield information on their name, URN, current presence, running services and assemblies. If a device or a connection for some reason is no longer discovered it changes colour indicating that a contingency may have arisen. Once the device or connection is rediscovered, the colour will change back indicating a reestablishment of the normal state.

1. Expanded device window. Within the device window, services running on the device are displayed. The device window features the name and URN of the device and a time indicating when it was last seen by the discovery manager. If a device has not been seen for a period of time, the window changes colour to indicate that the device and its services and assemblies are missing or otherwise incapable of being discovered. This allows for the user to take measures.
2. The name of the device. Names of services and assemblies are given at the same place in their respective windows.
3. Text indicating when the specific device was last discovered on the network.
4. The URN of a service. URNs of devices and assemblies are given at the same place in their respective windows.

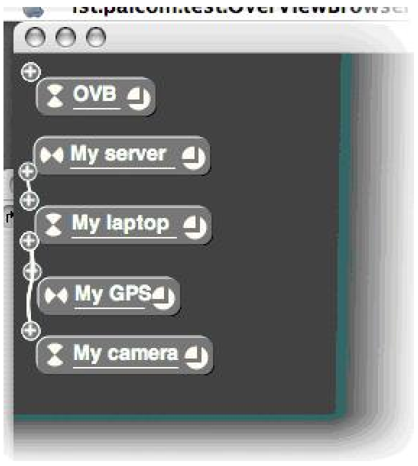
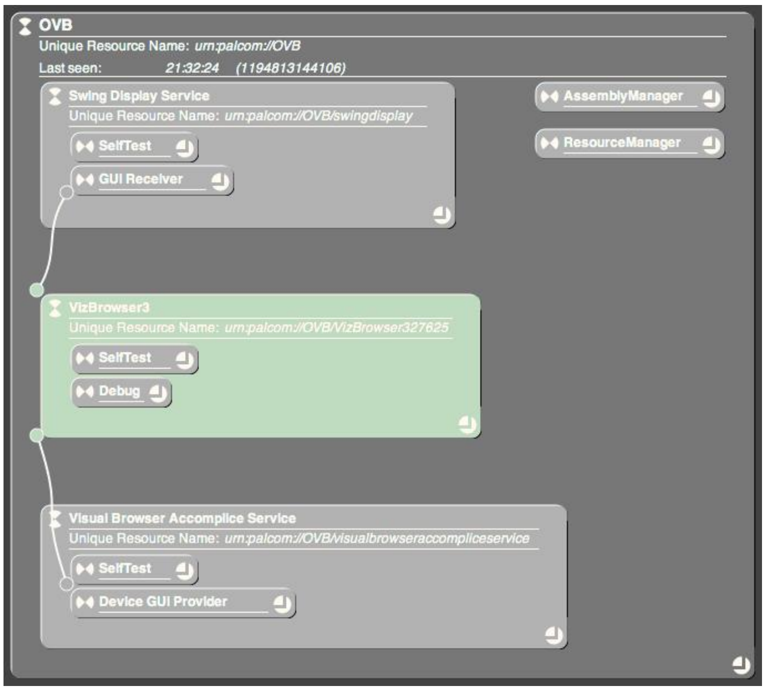


Figure 9: The image shows the initial state of OVB on a network with a series of devices available. Further, lines between some of the devices indicate that some of their services are cooperating in an assembly. To learn more about the assembly, the user can expand the different device windows.


OVB
Unique Resource Name: *urn:palcom://OVB*
Last seen: *21:32:24 (1194813144106)*

Swing Display Service
Unique Resource Name: *urn:palcom://OVB/swingdisplay*

VizBrowser3
Unique Resource Name: *urn:palcom://OVB/VizBrowser327625*

Visual Browser Accomplice Service
Unique Resource Name: *urn:palcom://OVB/visualbrowseraccompliceservice*

Figure 10: The topology of the Overview Browser device as shown by the OVBDevice itself. The device itself is depicted as the enclosing window encompassing five elements, three of which are the Swing Display Service, the Visual Browser Accomplice Service and an assembly reifying the viz3 assembly description, visualized as s kind of service and connected to the two services. Legend and visual detail descriptions can be found further down in the text.

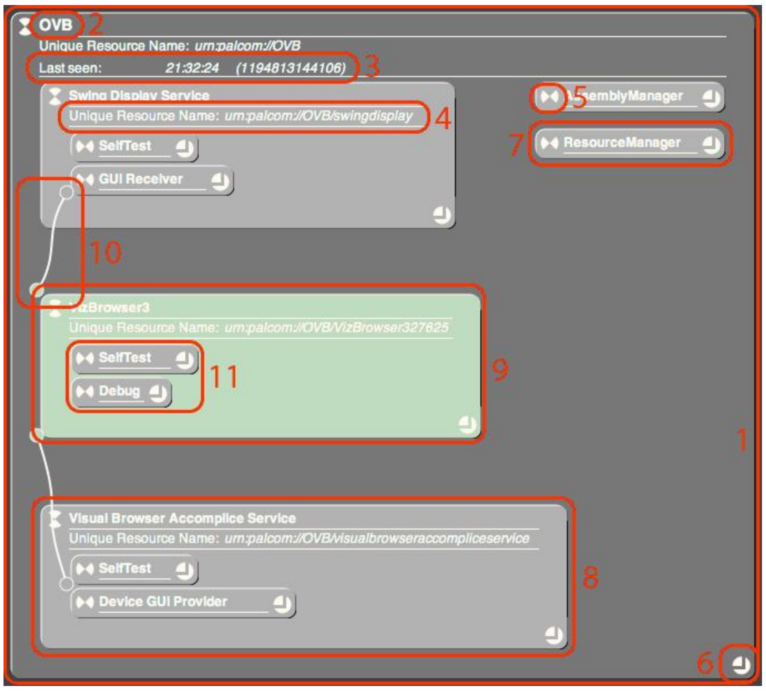
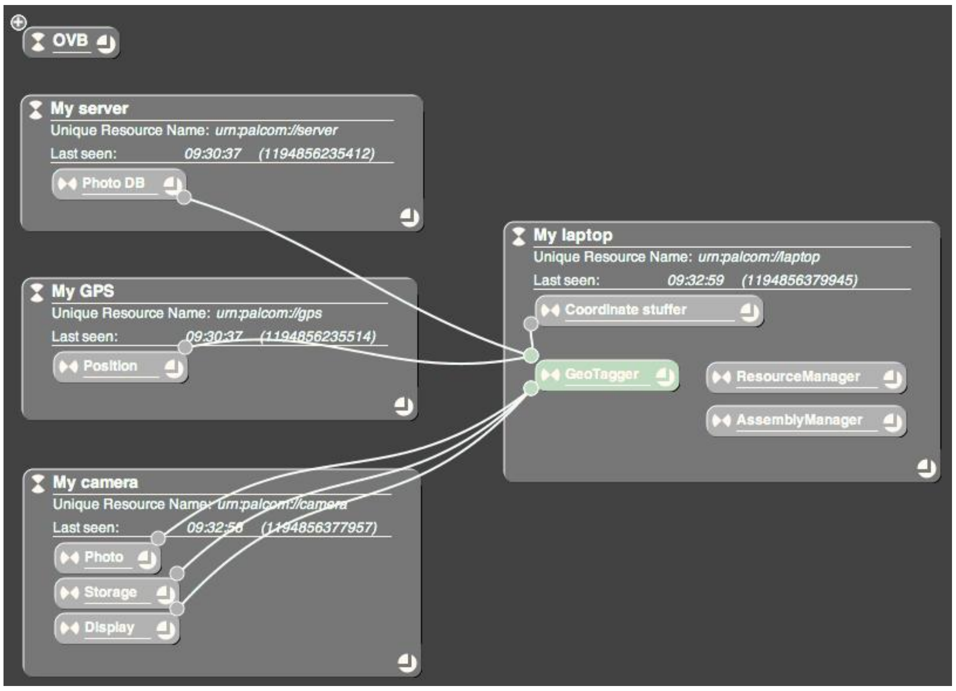

OVB
Unique Resource Name: urn:palcom://OVB
Last seen: 21:32:24 (1194813144106)
Swing Display Service
Unique Resource Name: urn:palcom://OVB/swingdisplay
SelfTest
GUI Receiver
AssemblyManager
ResourceManager
VizBrowser3
Unique Resource Name: urn:palcom://OVB/VizBrowser327625
SelfTest
Debug
Visual Browser Accomplice Service
Unique Resource Name: urn:palcom://OVB/visualbrowseraccompliceservice
SelfTest
Device GUI Provider
1
2
3
4
5
6
7
8
9
10
11

Figure 11: Legend showing the normative elements of the OVB. The encircled numbered elements are detailed below.

5. Minimize/maximize icon.
6. Resize icon.
7. Minimized service window. Services running on devices displayed within device windows are pr. default minimized.
8. Expanded service window. The service window features the name and URN of the service, along with the available groupings of control items.
9. Expanded assembly window. Assemblies are displayed as a differently coloured service, as this is how it acts from a discovery perspective.
10. A connection. As with device windows, connections change colour if they for some reason are suddenly no longer discovered.
11. Groupings of control items within an assembly indicative of its supplied functionality.

An example of a an inspection of an actual assembly, is the GeoTagger assembly shown in the figure below. The GeoTagger is a simple assembly consisting of a camera, a GPS device and a storage server. These devices can be connected by an assembly running on a laptop running an assembly manager and a service to annotate a picture with a GPS coordinate. Once the assembly has connected these devices and services a picture taken by the camera will be annotated with the current GPS position and stored on the server.



My server
Unique Resource Name: *urn:palcom://server*
Last seen: *09:30:37 (1194856235412)*

My GPS
Unique Resource Name: *urn:palcom://gps*
Last seen: *09:30:37 (1194856235514)*

My laptop
Unique Resource Name: *urn:palcom://laptop*
Last seen: *09:32:59 (1194856379945)*

My camera
Unique Resource Name: *urn:palcom://camera*
Last seen: *09:32:58 (1194856377957)*

Figure 12: An example inspection of the GeoTagger assembly. The assembly consists of four devices; a laptop, a GPS, a storage server and a camera.

7.2 PalCom Developer's Browser: User's guide

The PalCom Developer's Browser is a PalCom browser and a tool for editing assemblies, that has been developed as an Eclipse plugin. [2]

7.2.1 Installation

There are two ways of executing the PalCom Developer's Browser. One is as a stand-alone application that is pre-built in the PalCom tree, the other is to build from sources and run from an Eclipse workspace. The first option makes it easier to get started using the PalCom Developer's Browser, while the second option gives the possibility to modify the source code and develop the browser further.

Installing the stand-alone version In order to use the the stand-alone version, you need to put the PalCom binaries on your computer. This can be done by downloading the "Out of the Box" package *palbox.zip* from the PalCom download site, or by checking out the tree as described in Section 6.

Installing in a workspace For executing the PalCom Developer's Browser as an Eclipse plugin, you have to install Eclipse, as described in Section 6.2. The PalCom Developer's Browser has been tested with Eclipse version 3.3.

Check out and import projects as described in Section 6.3. That section describes how to check out the PalCom tree into one workspace, and import projects into another. Choose to import all projects below the directory `palcom/developer` (for following this guide, it is useful to import `palcom/doc` as well). After importing the projects, they have to be built, as described in Section 6.4.

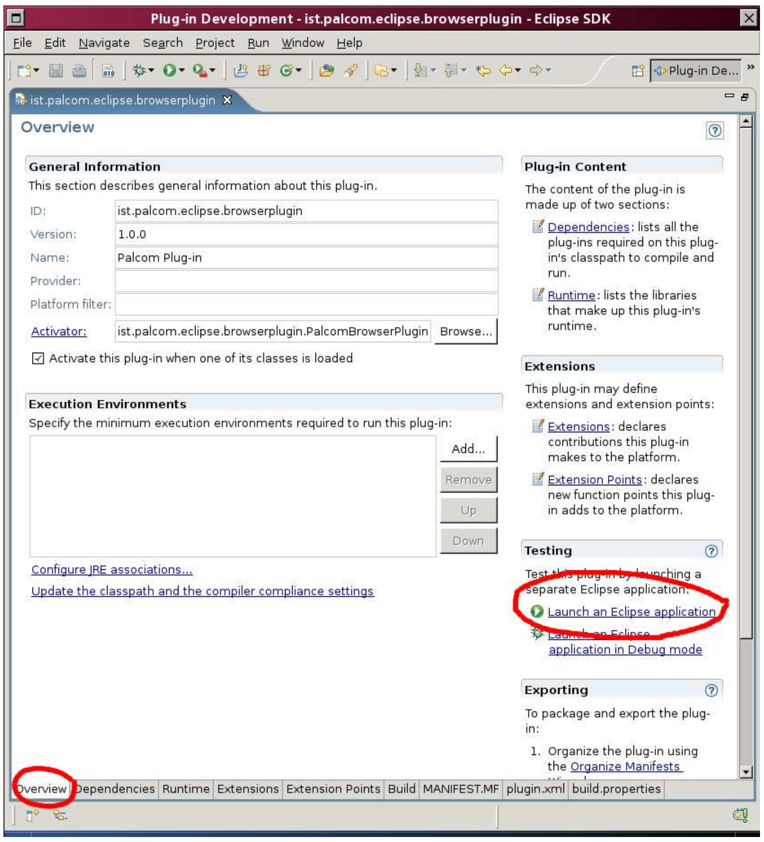
[2]The PalCom Developer's Browser is referred to as the *EclipseBrowser* in the PalCom source tree. This name in the source tree reflects that is *based on* Eclipse, not that a user of the browser will necessarily have to install Eclipse first.

7.2.2 Quick start

This section contains a very brief introduction to the PalCom Developer's Browser, and simulated PalCom devices. It first introduces the Eclipse application and the simulated devices, and then provides a walk-through of how to adapt and run an assembly.

Starting the stand-alone version If you have chosen to run the PalCom Developer's Browser as a stand-alone application, run the script `palcom/bin/pal-developers-browser` for starting the browser on your computer. Alternatively, you can find the right application for your OS below `palcom/lib/developers-browser`.

Starting the plug-in in Eclipse If, instead, you have chosen to run in a workspace, you start a new Eclipse instance including the plugin, from within Eclipse. One way to do this is to open "plugin.xml" in the Eclipse-Browser project, select the "Overview" pane, and do "Launch an Eclipse Application" (under "Testing"), as shown in Figure 13.


Plug-in Development - ist.palcom.eclipse.browserplugin - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Plug-in De...
ist.palcom.eclipse.browserplugin

Overview

General Information
This section describes general information about this plug-in.

ID:	ist.palcom.eclipse.browserplugin
Version:	1.0.0
Name:	Palcom Plug-in
Provider:	
Platform filter:	
Activator:	ist.palcom.eclipse.browserplugin.PalcomBrowserPlugin

Browse...

☑ Activate this plug-in when one of its classes is loaded

Execution Environments
Specify the minimum execution environments required to run this plug-in:

Add... Remove Up Down

Configure JRE associations...
Update the classpath and the compiler compliance settings

Plug-in Content
The content of the plug-in is made up of two sections:
- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- Runtime: lists the libraries that make up this plug-in's runtime.

Extensions
This plug-in may define extensions and extension points:
- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

Testing
Test this plug-in by launching a separate Eclipse application:
- Launch an Eclipse application
- Launch an Eclipse application in Debug mode

Exporting
To package and export the plug-in:
1. Organize the plug-in using the Organize Manifests

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

Figure 13: Launching an eclipse application from the plugin.xml

When you launch the new Eclipse instance, you are greeted by the Welcome screen, where you can access the Eclipse tutorials, etc. Click on "Workbench" to continue. This takes you to an empty workbench, with a Navigator, an Outline view, etc. Close all views (by clicking on the "x" in their title bars) except the Navigator.

Open a Palcom Browser view: From the menu, select Window→Show view→Other... and then open Palcom→Browser view.

Working in the PalCom Developer's Browser When the browser has been started, the basic work flow is the same, regardless of how it has been installed. Where there are differences, they will be pointed out specifically.

- *The browser view*

Dialog boxes will show, asking for device name and DeviceID for the browser. Enter values for them. (If you're alone on your network, it is safe to use the defaults. Otherwise, you must ensure that the DeviceID you enter is unique). The entered name and DeviceID are saved at the root of the Eclipse workspace, in the file ".EclipseBrowserDeviceSettings".

In the browser view, your Eclipse browser appears under "Devices", and you can browse its services, as seen in Figure 14.

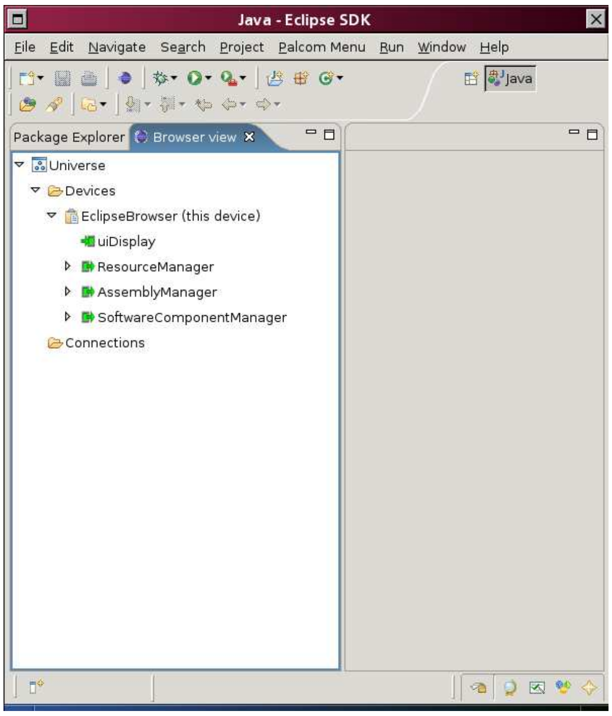


Figure 14: The palcom browser view showing the services of an EclipseBrowser device.

- *Creating projects and assemblies*

  To start editing assemblies, you first need to create an Eclipse project. Go to the Navigator. From the menu, do File→New→Project... In the wizard shown in Figure 15, select General→Project (click next) and enter a Project name. (Click Finish)

  Now you can create an assembly: File→New→File, Select the project you just created as the parent folder, and enter a file name, with the extension ".ass". E.g., "Test.ass" as shown in Figure 16. (click Finish).

- *Opening an assembly in the editor*

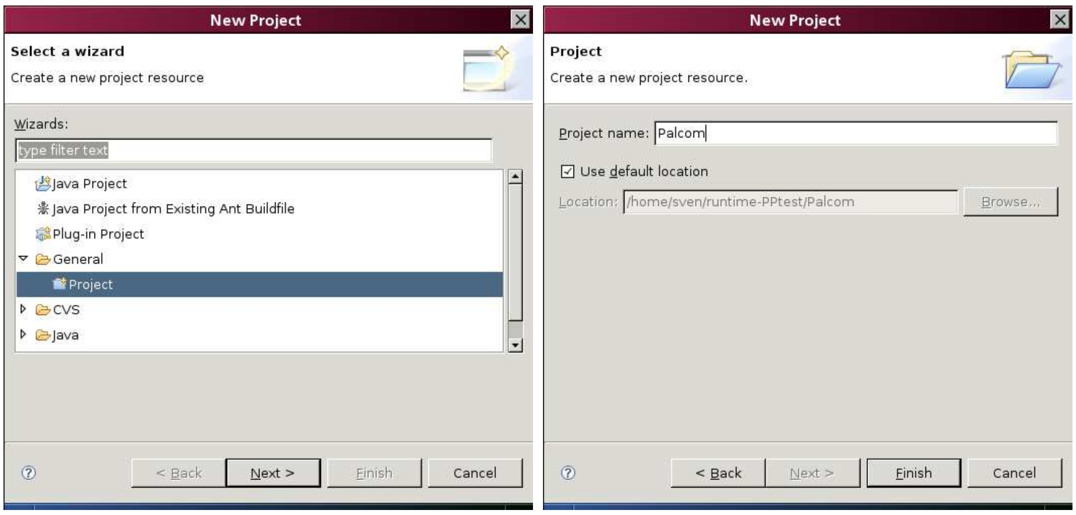
  When a new assembly file is created, an editor is opened, as shown in Figure 17. In this state, the actual file is still empty, but the editor has created a skeleton assembly, with a default name and one version (1.0) of it. If you close the editor without saving, the file will still be empty, but the next time that file is opened, a new skeleton will be created.

  One way of opening existing assemblies (or other files) from a project is using the Navigator view. If you closed it earlier, it is opened by Window→Show view→Navigator.

- *Customising the workbench layout*

  You can move views around on the workbench to get a layout you like by dragging them by their title bar. Then, if you run in a workspace, you can save that workbench layout ("perspective", in Eclipse lingo) by doing Window→Save Perspective As... and giving it a name (E.g., "PalCom"). The perspective can then be restored using Window→Open perspective.

Simulated PalCom devices In the PalCom DeviceFactory, there are a number of simulated PalCom devices that can be used for experimenting. Simulated devices can be created, started and stopped through

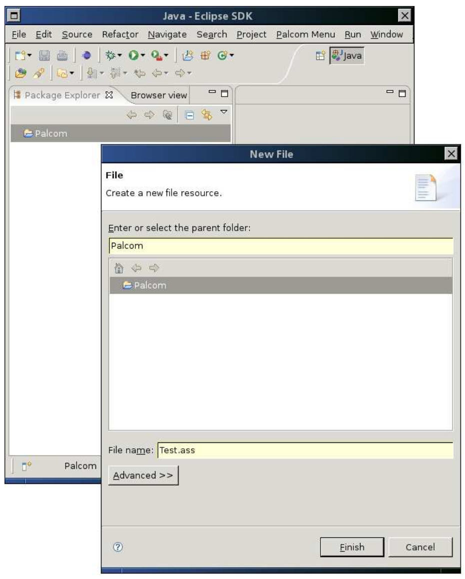


Project name: Palcom

☑ Use default location

Location: /home/sven/runtime-PPtest/Palcom

Figure 15: Creating a new project using the wizard.



Enter or select the parent folder: Palcom

File name: Test.ass

Figure 16: Creating a new assembly file using the wizard.

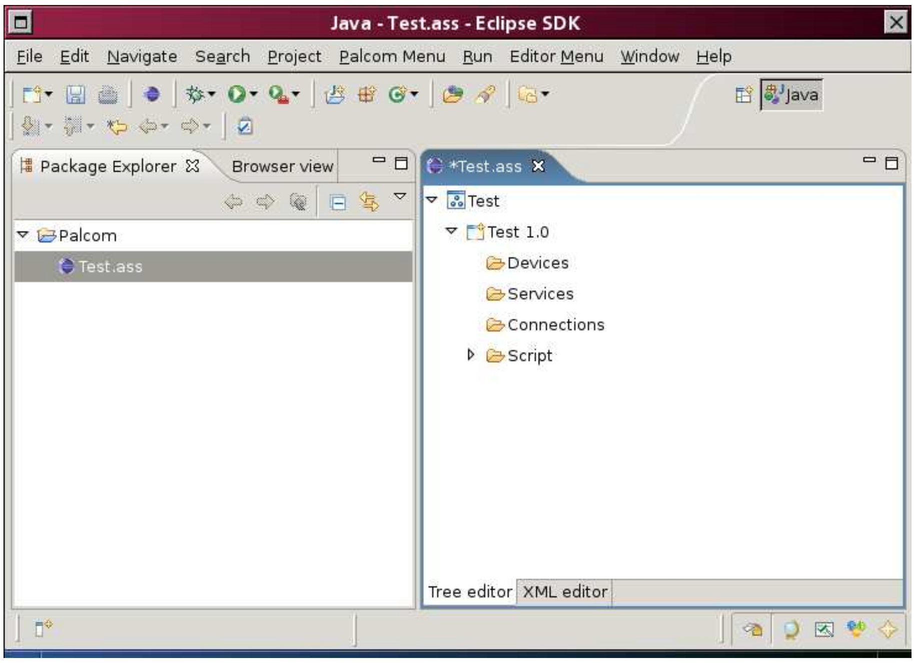


Figure 17: A newly created assembly, in the assembly editor. The figure shows the tree editor, where all the editing operations described in this document take place. There is also an XML editor, where the XML representation can be edited directly, but that is not recommended to the normal user. Unqualified uses of the term "assembly editor" refer to the tree editor.

the DeviceFactory. If running the stand-alone browser, start the DeviceFactory by running the script `palcom/bin/pal-device-factory`. If you run in a workspace, do like this: In the project named "factory", open factory→src→ist.palcom.simulated.device.factory, select DeviceFactory.java, and do "Run as→Java application" from the context menu.

New devices are created by doing "New device" from the context menu of the right pane. For instance to create a simulated GPS do:

1. New device
2. Device type: select ist.palcom.simulated.geotagger.GPSDeviceGUI (OK)
3. Enter a device name. E.g., "Joe's GPS" (OK)

To start a device, select it and do "Start device" from the context menu. Stop it in the same way. When you start the device, a user interface for the device is opened in a new window. It should also be discovered in the browser view.

Some of the simulated devices simulate the environment. E.g. the camera and GPS allow you to "walk around" by clicking and dragging in the grey area at the bottom of the window.

Assembly example In the directory `palcom/doc/tutorials/developers-browser`, there is an example assembly, the "GeoTagger.ass". If you run the browser in a workspace, you can import the `palcom/doc` project into your "developer" workspace, as described above. The "GeoTagger.ass" assembly connects a Camera, a GPS, a storage server, and an unbound service, "CoordinateStufferService".

We will now illustrate the assembly concepts by walking through this example. Please note that if you run in a workspace, you will have two Eclipse instances running. The first one is the one where you checked out and built the plug-in, the second one is the one actually running the browser and Assembly Editor plug-in. If nothing else is stated, the second Eclipse instance is considered.

1. *Loading the unbound service*

   The set of available unbound services can be changed manually. This is done by placing JAR files containing unbound services in a special directory "*workspace*/.EclipseBrowserDevice/SoftwareComponents", where *workspace* is the workspace directory of the Eclipse instance where the browser runs. (In order to see the path of your workspace directory, you can right-click on your project in the Navigator, and choose Properties and Info. The Location field shows where your workspace is.)

For preparing the CoordinateStufferService, first copy the JAR file `palcom-applications.jar` to the unbound services directory (the copying operation is best done outside Eclipse). The JAR file can be found under `palcom/lib/jars` or, if you run in a workspace, also in the *dev-lib* project under `jars/generated`.

Use the "Load Software Component" command from the context menu of the Browser view. In the first dialog box, scroll to `ist.palcom.simulated.geotagger.CoordinateStufferService`, select it and click OK. In the second dialog box, accept the default values and click OK. A new service "Coordinate stuffer" should appear on your browser device, in the Browser View.

2. *Get the example assembly*

   In the directory `palcom/doc/tutorials/developers-browser`, there is an example — the GeoTagger assembly. Copy this file into your assembly project.

   This can be done in many ways, for instance as follows:

   (a) Open the Navigator view
   (b) If you have imported the *doc* project (as described above): Drag the GeoTagger.ass from the folder *doc/tutorials/developers-browser* in the first Eclipse instance to your newly created project in the PalCom Developer's Browser. (NB! You have to drag the file to the actual Project node.)
   (c) Open the assembly in the editor by double-clicking.

3. *Create and start simulated devices*

   In order to run the assembly, you need to create and start simulated devices, as described above. You need a CameraDevice, a GPSDevice and a BackendServerDevice. Name them, for instance "Joe's Camera", "Joe's GPS", etc.

   After starting the devices, they should appear in the Browser view in your PalCom Developer's Browser.

4. *Rebind the assembly to your new devices*

   As the assembly was written for my (Sven's) devices, you have to change the declarations of devices and services to match your devices. Delete all device declarations, by right-clicking on them, choosing "Delete node", and clicking OK in the Confirm dialog box that pops up. Delete all service declarations in the same way. Then drag device nodes for the following four devices from the Browser view to the Devices section in the assembly:

   - Your CameraDevice.
   - Your GPSDevice.
   - Your BackendServerDevice.
   - Your EclipseBrowserDevice.

   Drag service nodes for the following six services from the Browser view to the Services section in the assembly:

   - The three services on the camera.
   - The Position service on the GPS.
   - The Coordinate stuffer service.
   - The Photo DB service on the BackendServerDevice.

   The assembly-internal names of the services will fit with the names used in the Connections section and in the assembly script.

5. *Start the assembly on your browser*

   Do "Start Assembly" from the context menu of the top node ("GeoTagger 1.0"). NB! The context menu of the icon, not the text.

   Now, the following should happen:

   - A new service, "GeoTagger", appears on the browser device
   - The connections declared in the assembly are established and shown under "Connections" in the Browser view

- Clicking the trigger button on the simulated camera should cause a tagged image to appear in the window of the simulated backing server.

Note that the assembly has a synthesized service, "Site pack". You can open a user interface by doing "Control" and then "Open UI" from its context menu. It also has a Debug service which can display status information about the Assembly. In the same way, you can explore other services, see "Remote control and user interfaces" below.

The assembly can be stopped by doing "Remove assemblies" from the context menu of the Browser View.

7.2.3 Usage

After the introductory example, we will now do an a bit more detailed tour of the facilities. The PalCom Developer's Browser has three parts, the Browser view, the Assembly Editor, and the XML view. They will now be presented.

When running in a workspace, the views are opened from the menu; Window→Show View→Other..., Palcom→Browser View or Palcom→XML view. When running the stand-alone version, the views are open from the beginning.

The editor is associated to the file extension ".ass", and opened whenever a ".ass" file is opened.

The browser view The browser shows a tree view of the network from the local device's point of view. I.e., all devices, services, and connections that it has found through discovery.

The "Universe" consists of the "Devices" tree showing all discovered devices and their services, and the list of Connections.

Browsing devices and services By unfolding a device or service node, its (sub)services are shown. Further, services having a ServiceDescription can be unfolded to show the ServiceDescription.

Manually setting up and closing connections To set up a connection, select two services (A customer and provider), and do "Connect" from the context menu.

To close a connection, select it in the "Connections" list, and do "Disconnect" from the context menu.

Remote control and user interfaces To remote control a (ControlProvider) service, select it and do "Control" from the context menu. This will set up a connection from the selected provider to the uiDisplay service on the browser. To open a user interface for a controlled service, select it and do "Open UI" from the context menu.

Control connections are closed as described above.

For services you are currently connected to, you can browse their commands in the browser view. "in" commands without parameters can be sent by double-clicking on them or doing "Invoke" from the context menu. Values of "out" command parameters are shown in the browser view. (For the content types "text/plain" and "image/jpeg").

Loading unbound services Unbound services[3] are loaded using the ServiceManager service on a device. First, an unbound service is registered (added to the list of available unbound services on that ServiceManager). Registered unbound services are then loaded using the load command. Registered unbound services are also loaded automatically when needed by an assembly. Thus, "Load Software Component" only has to be done the first time you load an unbound service on a device, or when you want to use it outside an assembly.

Registration basically maps a ServiceInstanceID to a fully qualified class name.

Unbound services are loaded onto the browser device by doing "Load Software Component" from the context menu.

"service name" is a fully qualified Java class name, e.g., "ist.palcom.simulated.geotagger.CoordinateStufferService"

The XML view For diagnostics, the XML representation of the currently selected node in the Browser view or Assembly Editor is shown. For instance, if a Service is selected in the BrowserView, the corresponding ServiceInfo is shown in the XML view.

[3]On the Java level, an unbound service is a subclass of ist.palcom.services.AbstractService. The class ist.palcom.service.manager.UnboundService is the glue that binds a service to a ServiceInstanceID and a device.

The assembly editor The assembly editor consists of two pages, one XML editor and one graphical editor for AssemblyInfos, where the latter one is the one opened by default, and typically used.

The XML editor is a normal text editor with syntax highlighting for XML.

The tree editor is a graphical, context-sensitive editor which relies on the discovered service descriptions to drive the editing. That is, currently, it is only possible to use services and commands that have been discovered (in the Browser view) in the assembly.

In the AST view, most editing operations are performed using drag-and-drop. The different editing operations will now be presented in some detail.

In addition to this, static-semantic errors in the AssemblyInfo are reported in the standard Eclipse view "Problems".

- *Creating a new Assembly*

  Create a new file (File→New→File) with the suffix ".ass". When creating or opening an empty file in the Assembly Editor, an empty assembly with a name matching the file name will be created upon changing to the AST view.

  Thus, to start over with an empty assembly, go to the XML editor pane, select and delete all text, and switch back to the AST pane. Alternatively, in the XML pane of the AssemblyEditor, click on the "Start over with an empty assembly" in the toolbar. (The only command there is). An empty XML represensation of an AssemblyInfo is created:

  ```
  ?xml version='1.0' encoding='ISO-8859-1' ?><!DOCTYPE AssemblyInfo SYSTEM "palcom.dtd">
  <AssemblyInfo name="AssemblyName">
    <DeviceDeclList />
    <ServiceDeclList />
    <ConnectionDeclList />
    <EventHandlerScript>
      <VariableList />
      <EventHandlerList />
    </EventHandlerScript>
  </AssemblyInfo>
  ```

- *Versioning*

  Assembly descriptors can contain several versions of the assembly, and new versions are created based on the existing ones. The general idea is that a new version is to be created for each (set of) change(s). In particular, once a version is *released* — i.e., has been made publically available — no further editing of that version is allowed. Instead, a new version should be created for subsequent changes.

- *Editing names*

  (most) Names can be edited by selecing them, changing the name, and hitting "Return". For devices and service declarations, the local name can be edited by clicking on the "top" declaration node. The other parts of the declaration are typically not edited, but can be by unfolding the declaration, as shown in Figure 18.

  As the presentation represents the AST, for some node types, there are "extra" levels in the tree, and you have to unfold one or two levels to get to the editable name. For instance, under "Connections", the first level is the ServiceUse node, and the next level is the editable Identifier. (In a more end-user oriented enviroment, the presentation would not show these intermediate nodes, but sometimes it is useful to view the XML representation of a node in the XML view, which is why the tree is presented as it is.)

- *Deleting nodes*

  Any declaration or clause can be removed by selecting it and doing "delete node" from the context menu. Deleting the list nodes ("Devices", "Services", "Connections") is currently possible, but a bad idea.

- *Declarations*

  – *Declaring a device:* Drag a device from the Browser view to the "Devices" node in the editor. In the same way, a device declaration can be re-bound to another DeviceID.

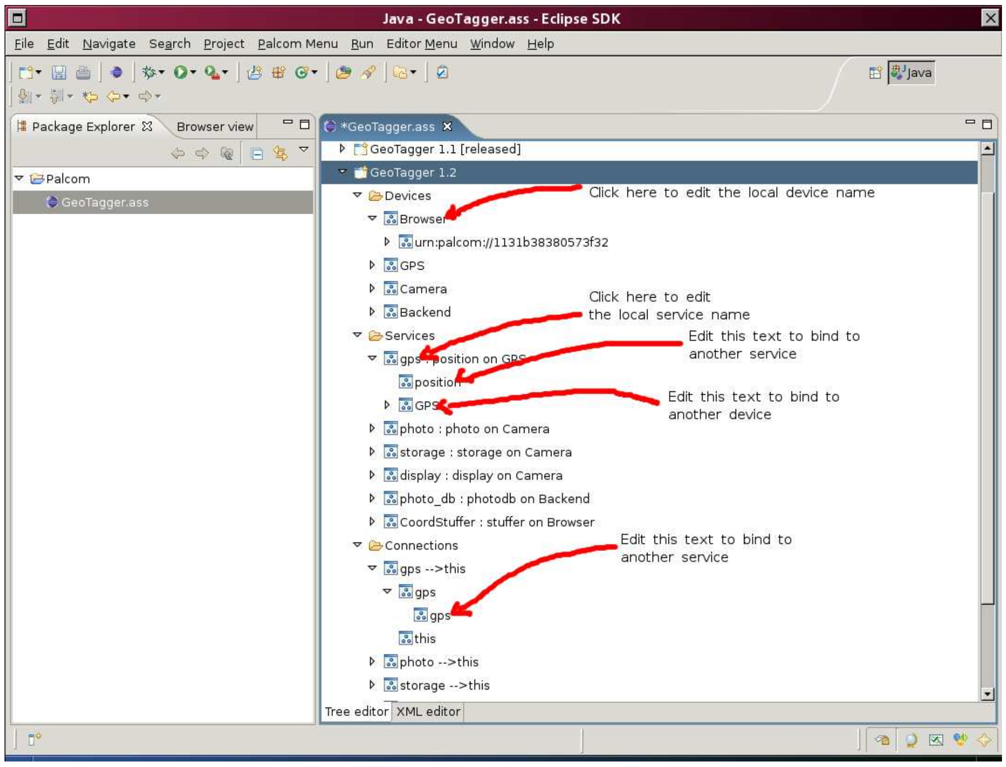

Java - GeoTagger.ass - Eclipse SDK
File Edit Navigate Search Project Palcom Menu Run Editor Menu Window Help
Package Explorer
Browser view
Palcom
GeoTagger.ass
*GeoTagger.ass
GeoTagger 1.1 [released]
GeoTagger 1.2
Devices
Browser
urn:palcom://1131b38380573f32
GPS
Camera
Backend
Services
gps : position on GPS
position
GPS
photo : photo on Camera
storage : storage on Camera
display : display on Camera
photo_db : photodb on Backend
CoordStuffer : stuffer on Browser
Connections
gps -->this
gps
gps
this
photo -->this
storage -->this
Tree editor
XML editor
Click here to edit the local device name
Click here to edit the local service name
Edit this text to bind to another service
Edit this text to bind to another device
Edit this text to bind to another service

Figure 18: Editing names in the assembly editor.

- *Declaring a service:* Drag a service from the Browser view to the "Services" node in the editor The service is given the same local name as the service name on the device. If you wish to change the local name, change it before declaring any connections, as subsequently declared connections will be bound to the local name.
- *Declaring a connection:* Drag a service from the Browser view to the "Connections" node in the editor to declare a connection from that service to the assembly. Drag a connection from "Connections" in the Browser view to the editor to declare a connection between any two services.

• *Script editing*

- *Variables* are declared by selecting the "Variables" node and doing "Add variable declaration" from the context menu. Enter type and variable name.
- *Eventhandler clauses* are declared by dragging a CommandInfo node (from a ServiceDescription) in the Browser view to the "Script" node in the Assembly editor.
- *Assignment statements* are added to an Eventhandler clause by selecting it and doing "Add assignment" from the context menu. Choose target and source variables from the dialogs.
- *SendMessage* statements are added by dragging a CommandInfo node from the Browser view to the EventHandler clause. If the command has parameters, dialogs appear allowing you to select parameter values.
  Alternatively,select an EventHandler clause and do "add Send Message" from the context menu.
- *Invoke statements* are the counterpart to SendMessage statements, on the assembly's own synthesized service, and are added a similar way; by dragging a CommandInfo from the ServiceDescription in the Assembly editor to the EventHandlerClause.
  Alternatively,select an EventHandler clause and do "add Invoke" from the context menu.

• *The synthesized service*

A synthesized service can be added to the assembly by selecting the top node (assembly name) and doing "Add Synthesized service" from the context menu.

Commands and Groups are added to the ServiceDescription by selecting it and doing "Add Command" or "Add Group" from the context menu. In the same way, commands are added to groups, and parameters to commands.

- *Running an Assembly*

  An assembly is run onto a device by passing its AssemblyInfo (as XML) to the load command of an AssemblyManager service. In the eclipse plugin, there is a short-cut for this: do "Start Assembly" from the context menu of the top node of the AssemblyInfo.

- *Stopping running assemblies*

  Running assemblies are stopped by sending the `unload_all` command to the AssemblyManager service. In the eclipse plugin, there is a short-cut: do "Remove assemblies" from the context menu of the Browser view.

7.2.4 Services on the EclipseBrowser device

The ability to run assemblies and load unbound services is associated with the concept of a PalCom browser. This is handled by two services, the AssemblyManager and the ServiceManager, which will now be described. Furthermore, there is a ResourceManager service, which is used by the AssemblyManager for looking up possible connections, etc.

(As mentioned above, in the Eclipse Browser/AssemblyEditor, there are short-cuts for some of the commands of these services).

AssemblyManager The AssemblyManager service is responsible for loading, executing, and unloading assemblies. Its ServiceDescription is as follows:

```
<ServiceDescription id="AssemblyManager">
  <GroupInfo>
    <CommandInfo id="list" direction="in" />
    <CommandInfo id="unload all" direction="in" />
  </GroupInfo>
  <CommandInfo id="alist" direction="out">
    <ParamInfo id="Assembly list" type="text/plain" dataRef="0" />
  </CommandInfo>
  <GroupInfo>
    <CommandInfo id="load" direction="in">
      <ParamInfo id="AssemblyInfo" type="text/xml" dataRef="0" />
    </CommandInfo>
  </GroupInfo>
</ServiceDescription>
```

The "list" in command lists (via the "alist" out command) the currently loaded assemblies.

The "unload all" in command stops and removes all running assemblies.

The "load" in command takes the XML representation of an AssemblyInfo, and creates and starts an Assembly instance.

ServiceManager The ServiceManager service is responsible for registering and loading unbound services. It keeps a record (on secondary storage) of the unbound services that have been registered. Its ServiceDescription is:

```
<ServiceDescription id="SoftwareComponentManager">
  <GroupInfo>
    <CommandInfo id="list_components" direction="in" />
  </GroupInfo>
  <CommandInfo id="component_list" direction="out">
    <ParamInfo id="Software Component list" type="text/plain" dataRef="0" />
  </CommandInfo>
  <GroupInfo>
    <CommandInfo id="load" direction="in">
      <ParamInfo id="URN" type="text/plain" dataRef="0" />
    </CommandInfo>
    <CommandInfo id="register" direction="in">
      <ParamInfo id="ClassName" type="text/plain" dataRef="0" />
      <ParamInfo id="URNSuffix" type="text/plain" dataRef="0" />
    </CommandInfo>
  </GroupInfo>
</ServiceDescription>
```

The "`list_components`" in command lists (via the "`component_list`" command) the set of registered software components. The list consists of the ServiceInstanceIDs the services will get when loaded.

The "load" in command loads and starts the service corresponding to a software component with a certain URL.

The "register" in command associates a fully qualified class name with an ServiceInstanceID.

(Doing "Load Software Component" from the context menu of the Browser view first registeres the unbound service and then loads it.)

ResourceManager To be written.

7.3 PalCom Commandline Assembler

The command line assembler is a utility for executing assembly scripts via a command line interface. The utility can be started with the `pal-dev-ass` script in `/developer/dev-bin`.

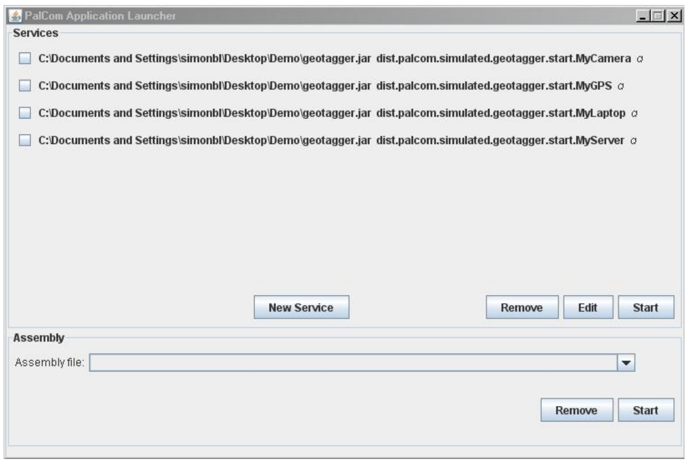
Type 'help' on the command line for more info.

7.4 PalCom Launcher

The PalCom launcher is a small utility providing a graphical user interface for:

1. Browsing and launching palcom services from jar files.
2. Using the command line assembler tool, see section 7.3

The code for the launcher is placed in `palcom/developer/tools/launcher` and can be started directly from Eclipse, or via the script `pal-dev-launcher`. Doing so will bring up the GUI shown in figure 19.



C:\Documents and Settings\simonbl\Desktop\Demo\geotagger.jar dist.palcom.simulated.geotagger.start.MyCamera
C:\Documents and Settings\simonbl\Desktop\Demo\geotagger.jar dist.palcom.simulated.geotagger.start.MyGPS
C:\Documents and Settings\simonbl\Desktop\Demo\geotagger.jar dist.palcom.simulated.geotagger.start.MyLaptop
C:\Documents and Settings\simonbl\Desktop\Demo\geotagger.jar dist.palcom.simulated.geotagger.start.MyServer

Figure 19: Screenshot of the launcher application

Pressing "New Service" will bring up a browse dialogue where the jar file containing the required service can be selected. After a jar file is selected the launcher prompts for a Main class to run from the jar file. To run one or more services, mark them in the service-list and press "Start". For convenience, your settings will be stored in a text file, "Bookmarks".

In the lower part of the GUI you can browse to and execute an assembly script (i.e. and ".ass" file).

7.5 `Pal-vm` Tool Suite

The Pal-VM Tool Suite is a collection of command line tools the support executing PalCom software on the `pal-vm`. An overview of the commands were given in section 6.5. The sections below provides additional details on these tools.

7.5.1 `Pal-vm`

The `pal-vm` is the PalCom Virtual Machine, which executes component files generated by the `pal-j`, `pal-asm`, and `pal-st` compilers, as described in section 4.4.4, and in the following sections.

The design of the virtual machine is described in detail in the PalCom Open Architecture deliverable [7].

`Pal-vm` Synopsis:
The `pal-vm` script can be invoked ad follows:

```
  pal-vm [options] [vm-arguments] <component.prc>

  Options:
    -h   help - print this text (use -help to see vm-arguments)
    -d   Use debug build
    -c   Use checked build
    -p   Use profiling build
```

The debug build provides vm light debugging support in the form of improved tracing of exceptions more detailed stack dumps than the defaults release build. Symbolic information such as the names of fields are kept at runtime. The aim of the debug build is to improve the debugging of the programs running on the vm.

The checked build has extensive runtime checks enabled, and will run very slowly, but is useful to debug the vm itself. The aim of the checked build is to catch internal errors in the vm to aid in debugging the vm. The checked build has a number of extra tracing options and an option to turn on constant-gc. When using constant-gc, the vm will run extremely slowly.

The profiling build enables support for GNU C++ profiling. Use the `gprof` program on the generated `gmon.out`. The specific `gprof` command to invoke is reported by `pal-vm` after execution.

`vm-arguments:`
The `vm-arguments` include the following:

```
  -v                         Verbose mode: display components loaded etc.
  -version
  --version                  Display pal-vm version and exit
  -trace-exceptions          Emit trace of exceptions thrown
  -trace-exceptions-after n  Emit trace of exceptions thrown, starting after n exceptions
  -trace-gc                  Emit trace of garbage collections
  -no-generations           Collect whole heap on every garbage collection
  -heap-size n[kM]          Set maximum size of heap (kilo- or megabytes) - default 8184k
  -help                     Display this help
  -cp <paths>               Specify paths to search for required components
  -base <path>              Specify path for ist.palcom.base.prc

Options only available in Debug build:
  -scanheap                 Emit info about the heap after each garbage collection

Options only available in Checked build:
  -trace                    Emit trace of bytecodes executed
  -trace-after n            Emit trace of bytecodes executed, starting after n bytecodes
  -trace-inter-dispatch     Emit tracing information for inter-language dispatching
  -trace-doesNotUnderstand  Emit tracing information for calls leading to doesNotUnderstand
  -constant-gc              GC constantly (very slow!)
  -preemption               Asynchronous preemption between processes
  -no-preemption            No asynchronous preemption (default)
```

`Pal-dev-vm` Synopsis:
The synopsis for `pal-dev-vm` is the same as for `pal-vm`.

7.5.2 `Pal-st`

The `pal-st`is the PalCom Smalltalk compiler, which generates PalCom bytecodes in PalCom binary components, which can be executed on the `pal-vm`.

For a description of the Smalltalk variant supported by `pal-st`, as well as the language interoperability mechanisms (FLIF) supported, refer to PalCom Open Architecture deliverable [7].

`Pal-st` Synopsis:

```
Usage: pal-st [options] <c1.st> ... <cn.st>
General options:
 -cp <path>                  : Specify component path
 -o <filenamename>           : Specify output file name
 -op <outputpath>            : Specify output directory
 -version or --version       : Display pal-st version, then quit
 -c <componentSpecFile>      : Specify component specification
 -v                          : Verbose output
 --help or --h               : Display this help

Options for manually specifying component information:
 -serviceMainClass <name>    : Specify class name to start as main service
 -requires <name:C.prc>      : Specify a required components

Advanced options:
 -import <C1:name>           : Specify an import (Beta legacy support)
 --disable-prefixes          : Disable prepending language-specific
                               prefix in generated method identifiers
 --generate-extern <language>: Generate a dummy interface
                               implementation for export to the
                               language (supported: java)
 --enable-component-interface-metainfo:
                               Generate FLIF-2 and deprecated pal-j
       component metainfo
```

`Pal-dev-st` Synopsis:
The synopsis for `pal-dev-st` is the same as for `pal-st`.

7.5.3 `Pal-j`

The `pal-j`is the PalCom Java compiler, which generates PalCom bytecodes in PalCom binary components, which can be executed on the `pal-vm`.

For a description of the Java variant (Pal-J) supported by `pal-j`, refer to PalCom Open Architecture deliverable [7].

`Pal-j` Synopsis:

```
Usage: pal-j[.bat] <options>
  Options:
      -c <component>           Mandatory component description file
      -verbose                 Output messages about what the compiler is doing
      -classpath <path>        Specify where to find user class files
      -sourcepath <path>       Specify where to find input source files
      -bootclasspath <path>    Override location of bootstrap class files
      -extdirs <dirs>          Override location of installed extensions
      -help                    Print a synopsis of standard options
      -version                 Print version information
```

`Pal-dev-j` Synopsis:
The synopsis for `pal-dev-j` is the same as for `pal-j`.

7.5.4 `Pal-asm`

The `pal-asm` is the PalCom bytecode assembler, which transforms textual PalCom bytecodes into PalCom binary components, which can be executed on the `pal-vm`.

For a description of the bytecodes supported by `pal-asm`, refer to PalCom Open Architecture deliverable [7].

`Pal-asm` Synopsis:

```
Usage: pal-asm[.bat] <options> <palcom bytecode assembler (.pra) file>
  Options:
    --prefix prefix
            Set default prefix
    --ignore-no-prefix
            Ignore lack of a prefix when assembling
    --version
            Display pal-asm version
    --compress
     -z
            Compress components with gzip
    --help
     -h     Display usage information
```

`Pal-dev-asm` Synopsis:

The synopsis for `pal-dev-asm` is the same as for `pal-asm`.

7.5.5 `Pal-dis`

The `pal-dis` is the PalCom bytecode disassembler, which transforms PalCom binary components into textual assembler files, which can ne edited and later re-assembled into component files using `pal-asm`.

`Pal-dis` Synopsis:

```
Usage: pal-dis[.bat] <options> <palcom component file>
  Options:
    --help
     -h       Display usage information
    --version Display pal-dis version
    --nocode
     -c       Do not print bytecodes of method bodies
    --ignoreprefix
              Ignore prefix meta information
```

`Pal-dev-dis` Synopsis:

The synopsis for `pal-dev-dis` is the same as for `pal-dis`.

7.5.6 `Pal-dep`

The `pal-dep` is the PalCom component dependency analyzer, which can be used to display the transitive closure of components a given component depends on (through the `requires` specifications of PCS files, see section 4.4.4 on page 20.

This is especially useful for determining the minimal set of component files to deploy for a device with small storage capabilities.

`Pal-dev-dep` Synopsis:

```
Usage: pal-dep <options> <palcom component (.prc) file>
  Options:
    --linebreaks
     -l
              Output on separate lines
     -cp path
              specify component path
    --version
              Display pal-dep version
    --help
     -h       Display usage information
```

`Pal-dev-dep` Synopsis:

The synopsis for `pal-dev-dep` is the same as for `pal-dep`.

A `Pal-vm` Component Specification File Format

This appendix describes the syntax of the component specification file.

A.1 File name convention

Component specification files should have names that end with the `.pcs` extension (for palcom component specification). This convention is not enforced by the compilers. The name `Component` without extension is often used in Java applications for historical reasons.

A.2 Grammar for Component Specification File

```
<component>       := "component" <name> <component-body>
<name>            := <identifier> ( "." <identifier> ) *
<component-body>  := "{" <feature> "}"
<feature>         := <class> | <mainclass> | <requires>
<class>           := "class" <name>
<mainclass>       := "mainclass" <name>
<requires>        := "requires" <string> "as" <name>
```

B `Pal-vm` Textual Assembler Reference

This appendix describes the `pal-vm` textual assembler, and the syntax for its input files.

B.1 File name convention

`Pal-vm` textual assembler files must have names that end with the `.pra` extension.

B.2 Assembly Language Syntax

The assembly language syntax is as follows, using EBNF [1].

```
<component>          ::= ( <metainfo> | <comment> | <class> )*

<metainfo>           ::= .metainfo <symbol> <symbol> <newline>
<comment>            ::= ; <not_newline>* <newline>

<class>              ::= <classname> <supername> <instance_or_class>*
<classname>          ::= .class <name> <newline>
<supername>          ::= .super <name> <newline>
<instance_or_class> ::= <instance_side> | <class_side> | <instance_method>
                      | <class_method> | <instance_field> | <class_field>
<instance_side>      ::= .instance_side <newline> { <newline> <member>* }
                         <newline>
<class_side>         ::= .class_side <newline> { <newline> <member>* }
                         <newline>
```

```
<member>            ::= <local_method> | <local_field>
<local_method>      ::= .method <symbol> <newline> <method_body> <newline>
<instance_method>   ::= .instance_method <symbol> <newline> <method_body>
                        <newline>
<class_method>      ::= .class_method <symbol> <newline> <method_body>
                        <newline>
<local_field>       ::= .field <symbol> <newline>
<instance_field>    ::= .instance_field <symbol> <newline>
<class_field>       ::= .class_field <symbol> <newline>

<method_body>       ::= { <newline> <metainfo>* <method_locals>
                          <method_stacksize> <bytecode>* }
<method_locals>     ::= .locals <number> <newline>
<method_stacksize>  ::= .maxstacksize <number> <newline>
<bytecode>          ::= <label>? <code> <argument>* <newline>
<code>              ::= halt | dup | push.local | push.argument | push.field
                      | push.block | push.constant | push.global | pop
                      | pop.local | pop.argument | pop.field | send
                      | super.send | return.local | return.non.local
                      | branch | branch.identical | swap | call
                      | branch.if.true | branch.if.false
<argument>          ::= <integer> | <symbol> | <string> | <array>
                      | <hashmap>
<integer>           ::= <number> | -<number>
```

```
<symbol> ::= <quoted_symbol> | <simple_symbol>
<quoted_symbol> ::= '#' <string>
<simple_symbol> ::= '#' <word>
<word> ::= <alphanum>+
<words> ::= <word> (' ' <word>)*
<alphanum> ::= <letter> | <digit> | '_'
<string> ::= '"' (""|<anychar>)* '"'
<label> ::= <string> ':'
<not_newline> ::= <anychar>*
<anychar> ::= <letter> | <digit> | ' ' | ...
<number> ::= <digit>+
<array> ::= ( <argument>* )
<hashmap> ::= { ( ( <symbol> | <integer> ) := <argument> )* }
```

The semantics attached to the metainfo parameters are described in Appendix "Metainformation" and Appendix "Reflection Data" in the PalCom Open Architecture Specification [7].

The exact types and numbers of arguments that are allowed for each bytecode are documented in Appendix "Bytecodes" of the PalCom Open Architecture Specification [7].

References

[1] ISO/IEC 14977:1996. Information technology – Syntactic metalanguage – Extended BNF. Technical report, ISO Standards, 1996.

[2] The eclipse project. `http://eclipse.org`.

[3] The gnu compiler collection. `http://gcc.gnu.org`.

[4] The palcom project download page. `http://www.ist-palcom.org/download`.

[5] The pomp project. `http://wiki.daimi.au.dk/som/the_pomp_project.wiki`.

[6] PalCom Project. Palcom external report 51: Deliverable 36 (2.14.1): Dissemination. Technical report, PalCom Project IST-002057, October 2006. `http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-36-[2.14.1]-dissemination.pdf`.

[7] PalCom Project. Palcom external report 69: Deliverable 54 (2.2.3): Open architecture. Technical report, PalCom Project IST-002057, December 2007. `http://www.ist-palcom.org/publications/deliverables/Deliverable-54-[2.2.3]-open-architecture.pdf`.

[8] Kari Schougaard and Ulrik P. Schultz. POMP – Pervasive Object Model Project. 9th Workshop on Mobile Object Systems (Resource Aware Computing), ECOOP 2003, `http://www.daimi.au.dk/~kari/publications/mos03.pdf`, 2003.

[9] The som project. `http://wiki.daimi.au.dk/som/_home.wiki`.

[10] Lund University. Jastadd open source java-based compiler compiler system. `http://jastadd.cs.lth.se/`.