Migrating Java Computations On the Palm Connected Organizer

- Persistent Java for the Palm -

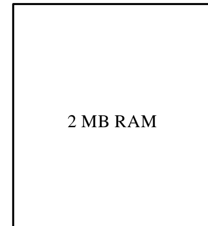
Daniel Schneider
University of Hamburg/Germany
(Sun Labs)

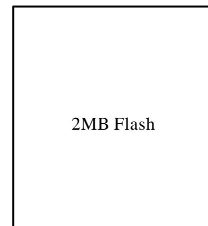
Migrating Java Computations On the Palm Connected Organizer

- Palm Basics
- Motivation
- Objective
- Store Structure
- External Data
- Summary

Palm Devices

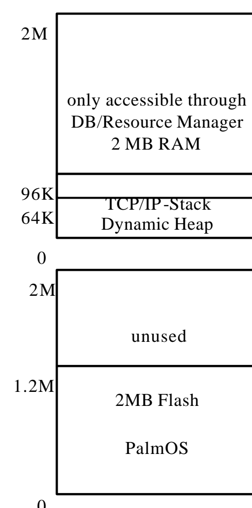
- **Palm III (x/e)**
  - Motorola Dragonball EZ (MC68EZ328)
  - 16.67 MHz
  - 2 MB RAM (upgradable to 8MB)
  - 2 MB Flash RAM
  - 1 card slot
  - IR port
- **Palm V**
  - same as III but slim design, Li-Ion battery
  - RAM upgrade not intended by 3com
  - no card slot
- **Palm IIV**
  - same V
  - includes a radio device to download mail and surf the web





Palm OS

- MacOS like operating system
- Mostly single threaded, event driven.
- The memory is divided into three parts
  - 96K of "dynamic" memory
  - 1.9MB of "static" memory
  - The "ROM" area: 2MB of Flash RAM
- The data and resource manager handles databases and resources.
  - Record databases
  - Resource databases
- Records/Resources must be locked when read from
- Records/Resources that are not locked can be moved by the OS



Motivation - why Palm

- The Palm computing market is growing rapidly
- Computing power of the PalmV compares to desktop systems 5 years ago
- Interesting computing model without secondary memory
- It opens new ways of computer use
  - ubiquitous computing
  - actual use of IR port
- It's cool

Motivation - why Migration

- Business processes can be modeled naturally using migrating computations
- Especially handheld computing offer new opportunities

Motivation - why Persistence

- PalmOS is a single tasking OS
- Application switches occur frequently as the organizer is used
- Application switches can occur at any point during program execution

➔ A Palm application must be able to handle persistence at any point

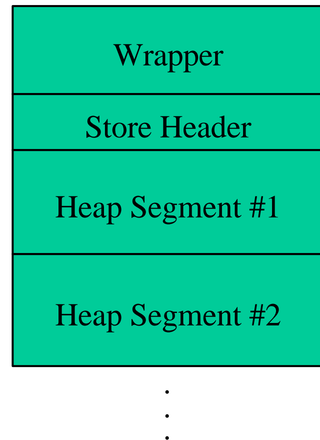
➔ An orthogonally persistent system provides this “for free”

Objective

- Allow the user to beam around “executing” Java computations captured in persistent stores.
- Unburden the Java programmer from having to handle persistence each time the user changes an application. Execution continues where it stopped, when the last change occured.
- Extend the limited heap size (<64 KB) to the size of free RAM (~1-2 MB).

Store Structure

- A store captures the complete state of an executing Java program
- It is a PalmOS resource DB of the type 'appl'
- It contains a wrapper that
  locates the VM and starts it with
  the store itself as the only parameter
- It contains a store header with
  bootstrap information and global variables
- It contains the segmented persistent heap:
  segments stored in records
  of the resource DB.
  All heap allocated objects are stored here.
- Computation is executed "in place"
- Suggested record size is ~64K (alap)



External Data

- External Data (*Externals*) is data that is not under control of the VM
- It often is an OS structure (File, Window, Database, Form)
- *Externals* have to be explicitly committed and closed
  when the store shuts down and reopened when the store restarts
- This is performed by the *ExternalManager*. An *External* has to register with the *ExternalManager* when it creates its external state.
- When the store is shut down all *Externals* are committed
  and closed by the *ExternalManager*
- When the store is restarted all *Externals* are recreated
  by the *ExternalManager*
- The use of synchronization guarantees that only a consistent state of an *Exteral* is made persistent
- We use the commit pattern as used by the Tycoon-2 system
  (can be downloaded at http://www.sts.tu-harburg.de/servlets/Tycoon)

Migrating Java Computations On the Palm Connected Organizer

- Allow the developer to model Workflows with a high degree of mobility (by both the user and the code)
- Offer orthogonal persistence with no cost for the application developer
- Allow the use of the whole free RAM (up to 8 MB)

- Next steps:
  - Serialize individual Objects
  - Serialize Threads to be beamed btw Palms in an Agent environment

Requirements for the GC

- exactness
- compaction
- multi segmented
- little memory overhead
- memory interface that minimizes the use of direct pointers (consistent/inconsistent)

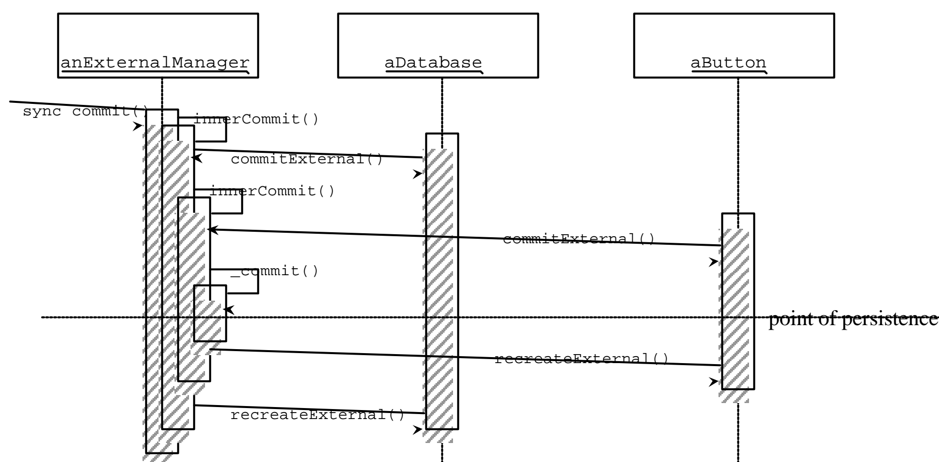
The *External* Interface

- A class whose instances deal with external data must implement the interface *External*
- The methods of *External* are exclusively invoked by the *ExternalManager*

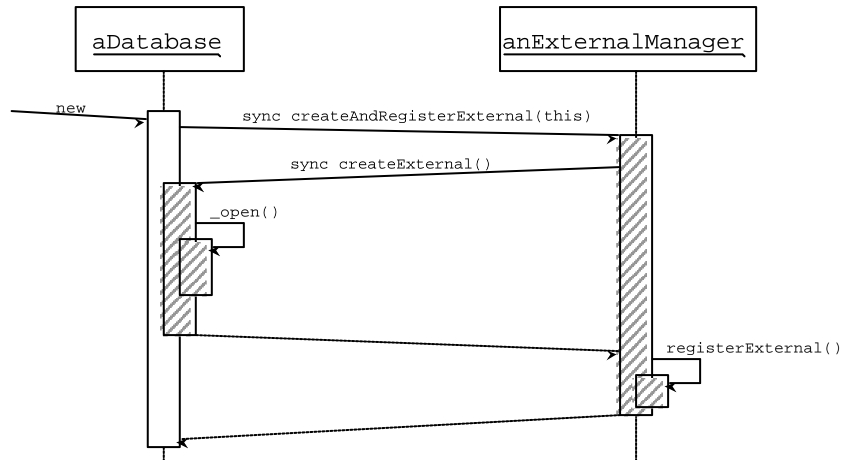
```
public interface External {
  public synchronized void createExternal();
  public synchronized void commitExternal();
  public synchronized void recreateExternal();
  public synchronized void destroyExternal();
}
```

```
public class ExternalManager {
  public synchronized void
    createAndRegisterExternal(External res);
  public synchronized void
    deregisterAndDestroyExternal(External
res);
  public synchronized void commit();
}
```

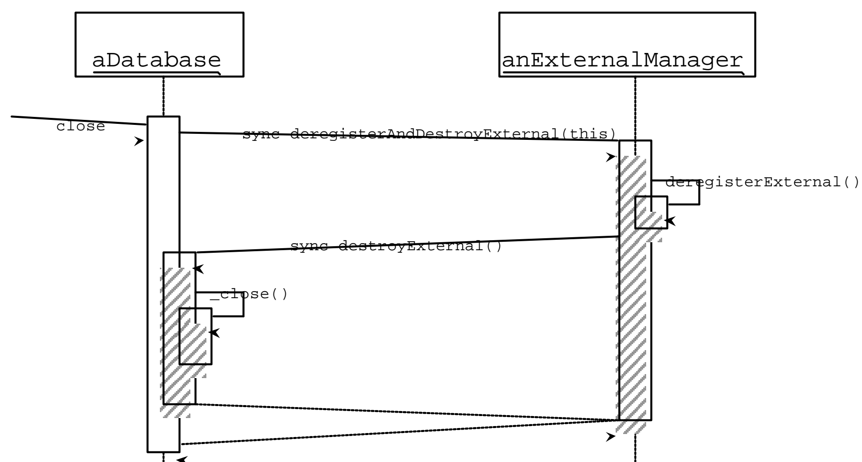

Committing the *ExternalManager*



Registering with the ExternalManager



Degistering from the ExternalManager



Why all this "synchronized"?

- because a commit can happen any second

```
Example 1:

public class Database {
  public Database(int someId){
    ExternalManager.getExternalManager().registerExternal(this);
---------------------------------------------------------------- point of persistence
    open(someId);
  }
  ...
}

Example 2:

public class Database {
  public void close{
    ExternalManager.getExternalManager().deregisterExternal(this);
---------------------------------------------------------------- point of persistence
    close(someId);
  }
  ...
}
```


Persistent System Classes HowTo #1

- Make the class implement the *External* interface
- Introduce variables to hold the persistent state
- Move pertinent code that puts the *External* into a state where external state matters to *createExternal()*
- Move code that puts the *External* into a state where external state no longer matters to *destroyExternal()*
- Implement *commitExternal()* to internalize the external state
- Implement *recreateExternal()* to reestablish the external state from the internal

```
• public class Database implements External

• private int typeID = 0;
  private int creatorID = 0;
  private int mode = 0;
• public void createExternal(){
    dbRef = _open(typeID, creatorID, mode);
  }

• public void destroyExternal(){
    _close();
    dbRef = 0;
  }

• public void commitExternal(){
      // nothing to be done in Database
  }

• public void recreateExternal(){
    dbRef = _open(typeID, creatorID, mode);
  }
```

Persistent System Classes HowTo #2

- when you want to call *createExternal()* call *createAndRegisterExternal(this)* instead.
- when you want to call *destroyExternal()* call *deregisterAndDestroyExternal(this)* instead.

```
• public Database(int typeID,
                  int creatorID, int mode){
      this.typeID = typeID;
      this.creatorID = creatorID;
      this.mode = mode;

      ExternalManager.getExternalManager()
          .createAndRegisterExternal(this);
  }
• public void close() {
    ExternalManager.getExternalManager()
      .deregisterAndDestroyExternal(this);
  }
```