JTRES 2010
19-21 August,
Prague,
Czech Republic

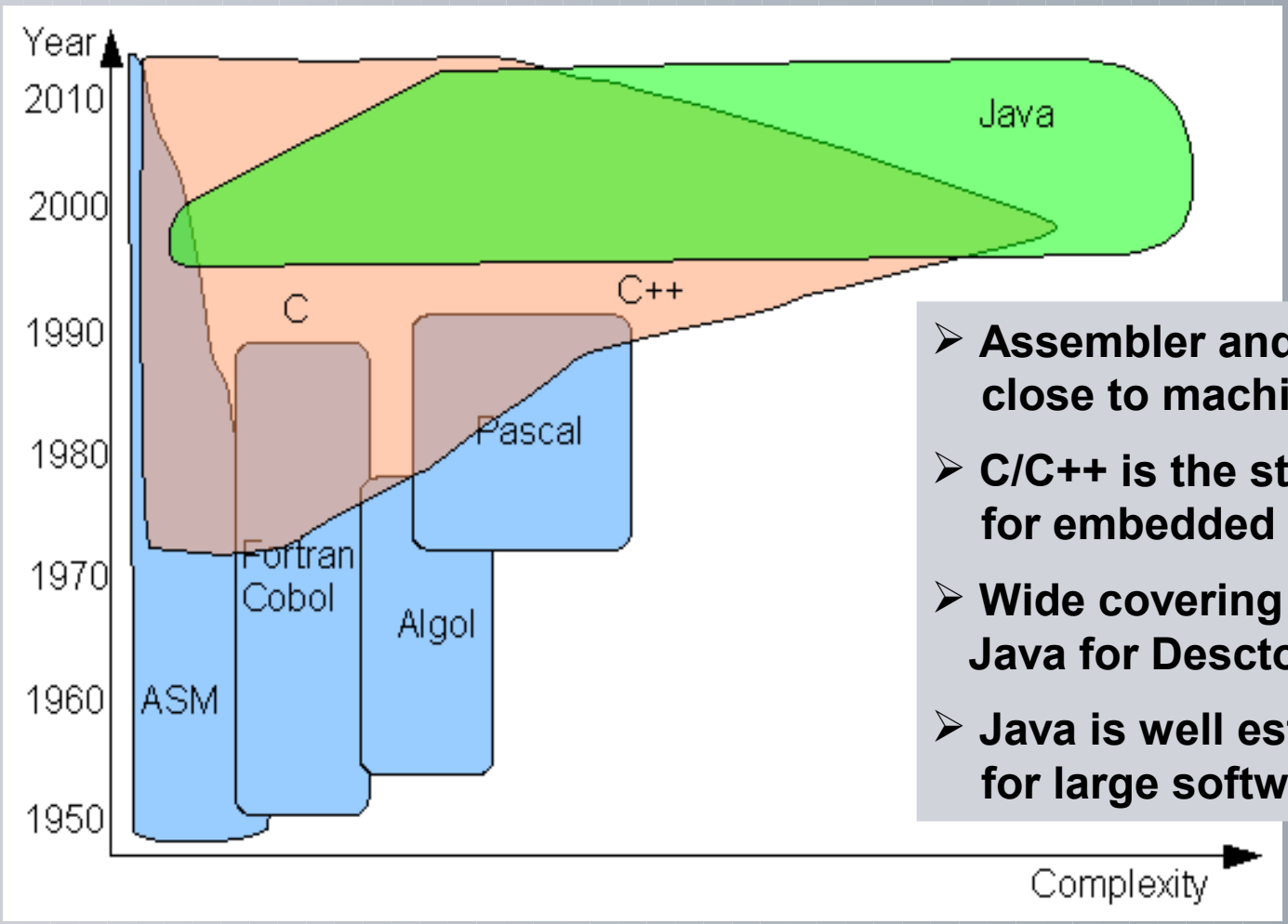
SIEMENS

# Java2C
## Developing in Java, Deployment in C

**Hartmut Schorrig & Thomas Henties, Siemens AG**

# Spread and Complexity of Programming Languages

SIEMENS




- **Assembler and C are close to machine code.**
- **C/C++ is the standard language for embedded applications.**
- **Wide covering of C++ and Java for Desctop-Applications.**
- **Java is well established for large software projects.**

# Java compared to C and C++

## C

- An old language, but close to machine code.
- Large variety of libraries, frameworks, styles for basics already available.
- Gives a lot freedom to programmers, but fatal errors are hardly avoidable.

➢ Intensive test necessary

## C++

- All disadvantages of C.
- Complex high-level-language-like constructs (templates
- It is a wolf in the sheep's clothes.

## Java

- Robust, Object Oriented language
- Safe object references
- Easy, safe synchronization
  Machine/Architecture independent
- **Compile 'n run:** **Mistakes cause meaningful error-messages.**

**Java code**

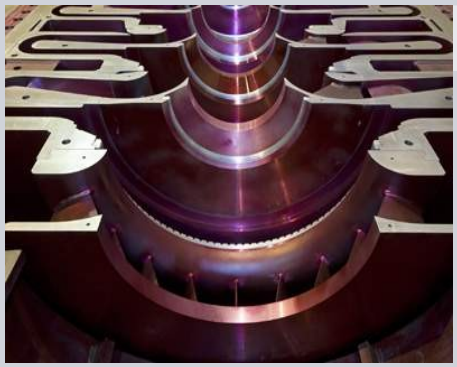
➔ easy to understand
➔ precise semantics
➔ machine independent

SIEMENS

# But C/C++ is still widely spread in Industry !







SIEMENS

# Java2C helps to bridge the Gap

**Java2C enables:**

- The use Java for design and implementation
- Automatic transformation into C
- Easy integration with
  - existing software
  - tool chains and
  - platforms.

**Real World Development**

- Classic C/C++ tool chain for embedded applications
  - C/C++-Compiler
  - Linker
  - Debugger
- Core functionality of hardware in focus
- Experienced developers available

**Software-Design in Java**

- Language avoids errors
- Good tool support
- Supports object oriented design

# A 16-bit-fixpoint code-example with structured variables in class namespace

SIEMENS

**gensrc_c\PosCtrl\PID_Controller.c**

```
/**the step-method, called one-time per cycle-time
to...*/
int16 calculate_PID_controller_F(PID_controller_s*
ythis, int16 input, ThCxt* _thCxt)
{ STACKTRC_TENTRY("calculate_PID_controller_F");
  { int32 intgValNew;
    int32 out;
    intgValNew = ythis->intgVal + (input * ythis->kI);
    out = (intgValNew >> 16) + ((ythis->kP * input) >>
6);

    /*The out may be overdriven, but it can be
limited...*/
    if(out > 0x7fff)
    { out = 0x7fff; }
    else if(out < -0x8000)
    { out = -0x8000;}

    /*Test if the integral value is overdriven. */
    if(((ythis->intgVal > 0 && input > 0) || (ythis-
>intgVal < 0 && input < 0)) && ((intgValNew ^ ythis-
>intgVal) & 0x80000000) == 0x80000000)
    {
      /*The sign of new value is changed, ...,*/
      throw_s0Jc(ident_RuntimeExceptionJc, "integral
value overdriven", 0, &_thCxt->stacktraceThreadContext
, __LINE__);
    }
    else
    { ythis->intgVal = intgValNew;
    }
    { STACKTRC_LEAVE;
      return (int16)(out);
    }
  }
  STACKTRC_LEAVE;
}
```

**org/vishia/exampleJava2C/java4c/PID_Controller.java**

```
/**the step-method, called one-time per cycle-time to...
 * The integral value is incremented by input. The result...
 */
public short calculate(short input)
{
  int intgValNew = intgVal + (input * kI);
  int out = (intgValNew >> 16) + ((kP * input)>>6);

  /*The out may be overdriven, but it can be limited...*/
  if(out > 0x7FFF)
  { out = 0x7FFF;
  }
  else if(out < -0x8000)
  { out = -0x8000;
  }

  /*Test if the integral value is overdriven. */
  if( (  (intgVal > 0 && input > 0)
      || (intgVal < 0 && input < 0)
      )
    &&((intgValNew ^ intgVal ) & 0x80000000 ) == 0x80000000
    )
  { /* The sign of new value is changed, but the ...*/
    throw new RuntimeException("integral value overdriven");
  }
  else
  { intgVal = intgValNew; //the integral value is valid.
  }

  return (short)(out);
}
```

SIEMENS

# An interface in Java becomes a direct call in C

**org/vishia/exampleJava2C/java4c/MainController.java**

```
/**The instance of set-value generator.
  * The instance is referenced with an interface
  * in Java, but in C there is direct access to
  * @java2c=instanceType:"SetValueGenerator".*/
private final SetValueGenerator_ifc
                mySetValueGenerator;

...
boolean isSet =  mySetValueGenerator.
                              setTarget(targetValue);
```

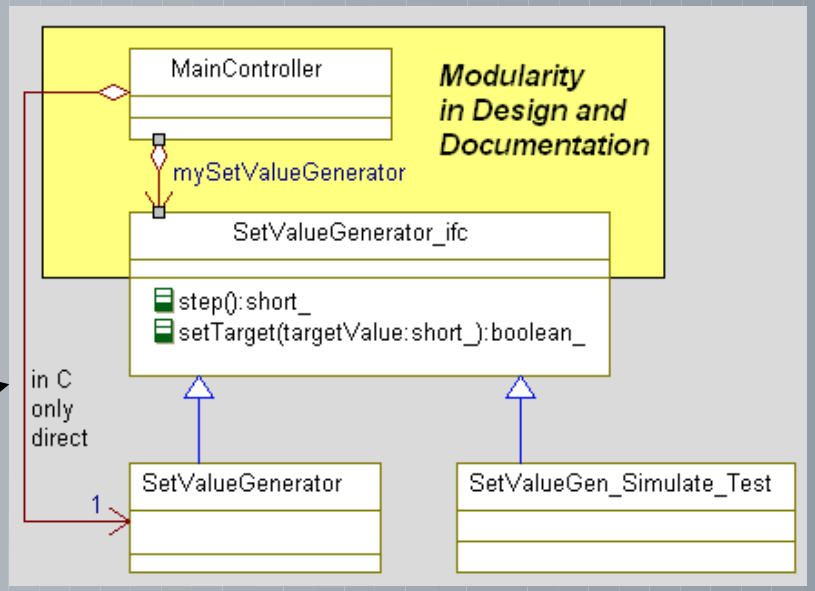
**gensrc_c\PosCtrl\MainController.h**

```
/*@CLASS_C MainController @@@@@*/

typedef struct MainController_t
{ ...
  struct SetValueGenerator_ifc_t*
  mySetValueGenerator;
  ...
} MainController_s;
```

**gensrc_c\PosCtrl\MainController.c**

```
bool isSet;
isSet = setTarget_i_SetValueGenerator_F(
  &((ythis->mySetValueGenerator)->base.object)
  , targetValue, _thCxt);
```



SIEMENS

# Using interface / overridden method in C

**gensrc_c\PosCtrl\WaySensor.h**

```
typedef int32 MT_getWay_WaySensor(ObjectJc* ithis, ThCxt* _thCxt);
...
typedef struct Mtbl_WaySensor_t
{ MtblHeadJc head;
  MT_getWay_WaySensor* getWay;
  Mtbl_ObjectJc ObjectJc;
} Mtbl_WaySensor;
```

**gensrc_c\simPc\SimPc.c**

```
/**J2C: Reflections and Method-table **/
const MtblDef_waySensor1_SimPc mtblwaySensor1_SimPc = {
...
, { { sign_Mtbl_WaySensor//J2C: Head of methodtable.
    , (struct Size_Mtbl_t*)((1 +2) * sizeof(void*))     }
  , getWay_waySensor1_SimPc_F    ...
```

**gensrc_c\PosCtrl\MainController.c**

```
typedef struct WaySensorMTB_t
{ struct Mtbl_WaySensor_t const* mtbl;
  struct WaySensor_t* ref;
} WaySensorMTB;

/**It is the main loop of the fast controller thread.*/
void run_WayCtrlThread_MainController(ObjectJc* ithis, ThCxt* _thCxt)
{ ...
  WaySensorMTB way1Sensor;
  SETMTBJc(way1Sensor, REFJc(ythis->outer->way1Sensor), WaySensor);
  ...
  while(true)
  { …
    xWay = way1Sensor.mtbl->getWay(
       &(( (way1Sensor.ref))->base.object), _thCxt);
  …
  }
```

**The implementing method**

**org/vishia/exampleJava2C/java4c/WaySensor.java**

```
/**This interface is the access to Hardware. */
public interface WaySensor
{  /**gets a actual way */
  public int getWay();
}
```

**org/vishia/exampleJava2C/simPc/SimPc.java**

```
final WaySensor waySensor1 = new WaySensor()
{ public int getWay()
  { return (int)(way1);
  }
};
```

**org/vishia/exampleJava2C/java4c/MainController.java**

```
/**Aggregation to a way sensor for input for control. */
  private final WaySensor way1Sensor, way2Sensor;
…
@Override public final void run()
{
  /**Internal variable to hold the reference to the interface,
   * @java2c=dynamic-call. In C the reference to the method-tabl
   * will be gotten one time here in startup-phase,
   * to relieve the time-critical calc-time in the loop. */
  final WaySensor way1Sensor = MainController.this.way1Sensor;
…
  while(true){
    /**Wait until cycle, it is a wait/notify from interrupt */
    waitCycleOrganizer1.waitCycle();
    /**reads the inputs from Hardware, it is in units 1 micromet
    int xWay = way1Sensor.getWay();
    …
  }
```

## Safety concept of method-tables for dynamic overrideable calls

SIEMENS

# Method pointers are never stored between data:

- Unintended side effects may corrupt data.
- Data can be tested using significance checks.
- Unlike in C++, the correctness of executed machine instruction is tested.
  - A method table is stored only in const data range (maybe write protected).
  - The pointer to the method table is only hold in stack – using the MTBL-struct.
  - To get the pointer to the method table of a given data instance, reflection is used, and three significance checks were done.
    Some more calculation time, but it is safe!

# Safe and fast algorithm to get the method table pointer

SIEMENS

```
MtblHeadJc const* getMtbl_ObjectJc(ObjectJc const* ythis
  , char const* sign)
{ MtblHeadJc const* head = null;

  ClassJc const* reflection;
  STACKTRC_ENTRY("getMtbl_ObjectJc");
   ASSERT(ythis->ownAddress == ythis);
  reflection = ythis->reflectionClass;
  if( reflection != null) {
    ASSERT(reflection->object.reflectionClass ==
           &reflection_ClassJc);
    head = ythis->reflectionClass->mtbl;
    if(head != null)//nullpointer  possible.
    {
      while(  head->sign != sign
           && head->sign != signEnd_Mtbl_ObjectJc
           )
      { int sizeTable = (int)head->sizeTable;
        ASSERT(sizeTable >0 && sizeTable < (302 * sizeof(void*)));
        //The next part of method table is found after the current.
        head = (MtblHeadJc const*)( (MemUnit*)head + sizeTable );
      }
      if(head->sign == signEnd_Mtbl_ObjectJc){
        THROW_s0(ClassCastException, "baseclass not found", (int)sign);
        head = null;  //return null admissible.
  } } }
  STACKTRC_LEAVE; return head;
```

*Instance contains its own address – protection against copy*

*Reflection is prooved to be valid*

*Searching is limited to a valid interval*

*Fast algorithm Short tables.*

SIEMENS

## Package-replacement and translation control

**Java package structure: java/util/.. or org/vishia/..
is replaced by special conventions in C:**

- Basic packages are supplied in C in the CRuntimeJavalike (Jc...)
- Some special classes in Java for C-functionality (bridgeC)
- Some packages: vishia/util are pre-translated and placed in J1c...
- User functionality in special directories
- Not everything has to be translated, usage of stc-files with structure information of classes for translation.

```
class SimpleDateFormat;
nameC=SimpleDateFormatJc_s; header=Jc/DateJc.h;
argIdent=Dt; extends java/lang/Object
{
  fields {
   int %.d DATE_FIELD;
  }
  methods {
    ctorO-_$, mode=static: SimpleDateFormat *..
return();
```

SIEMENS

## malloc/free (new/delete) is inhibited

- All blocks have the same size, 2048 Bytes.
- Blocks may consist of smaller *nodes*, which are freed and reused internally.
- Larger data structures may cover several blocks.

## Interruptible Garbage Collection is provided

- Works similar to a reference count Garbage Collection.
- Backward references are kept  for all references.
  - For safety sake when native code is involved.
  - For debugging purposes.
- Only entire blocks will be freed.

## More than one BlockHeap is supported

- At least one BlockHeap per thread or per module.

SIEMENS

## Conclusions

### C

- is well established and
- still the standard for proprietary and closed-hardware programming.

### Java

- has plenty of well known advantages.

### Java2C

- is a utilizable tool to consolidate Java with traditional C environments.
- works with Realtime-Java and Standard-Java.
- is ready to be spread in the world.

SIEMENS

# Questions & Answers

**www.vishia.org/Java2C/**

## Memory architecture

**Java:**
- Always dynamic,
- VM holds memory, Garbage-Collector
- Any instance with its own memory portion
- Dynamic data usage prevents data conflicts
- Large complex applications should be supported

- Allocation in Java at startup (constructor, final objects possible
- Re-using of data-container are not usual and not recomended, but possible => for C style.
- Annotations for Stack-instances etc.

**C:**
- Fix static areas of data,
- May be memory-address-bounded
- Embedded struct typically
- Short-living data in stack area
- Dynamic data sometimes inhibited
- Application are limited in quantity of parts often

```
public class MainController
{ …
  /**Embedded Composition a log message output to file. */
  public final LogMessageFile logMsgFileCtrlValues =
    new LogMessageFile("testLog/ctrl$MMdd_HHmmss$.log", 10
    , 1, msgDispatcher.getSharedFreeEntries());
```

```
/* @java2c=stackInstance, fixStringBuffer.
 * The StringBuffer is created in C in the stack
 */
final StringBuffer errorBuffer
  = new StringBuffer(100);
```

```
typedef struct MainController_t
{ ...
  LogMessageFile_MSG_s logMsgFileCtrlValues;
}

struct MainController_t* ctorO_MainController(ObjectJc* othis, …
{ ...
  init_ObjectJc(&(ythis->logMsgFileGC.base.object), sizeof(ythis->logMsgFileGC), 0);
  ctorO_LogMessageFile_MSG(/*static*/&(ythis->logMsgFileGC.base.object),
    sO_StringJc("testLog/GC.log"), 0, 0, getSharedFreeEntries_MsgDispatcher_MSG(&
    (ythis->msgDispatcher), _thCxt), _thCxt);
```

SIEMENS

# Java Advantages compared to C and C++

**Robust language**
- Designed to catch many errors at compile time
- Simple; easy to learn
- Intrinsic exception handling

**Object Oriented**
- Clean modularity
- Easy extensibility
- Information hiding is language concept

**Safe object references**
- Can't generate GeneralProtection Fault or Segmentation Fault
- Array bounds are checked (at compile-time when possible)

**Easy, safe synchronization**
- Uses "monitor" concept
- Threads as first-class constructs

**Machine/Architecture independent**
- Easy portable
- Tons of standardized packages

Java code
→ easy to understand
→ precise semantics
→ machine independent