A Model for Reuse and Optimization of Embedded Software Components

Mikael Åkerholm, Joakim Fröberg, Kristian Sandström, Ivica Crnkovic
MRTC, Mälardalen University
Västerås, Sweden
mikael.akerholm@mdh.se

Abstract. *In software engineering for embedded systems generic reusable software components must often be discarded in favor of using resource optimized solutions.*

In this paper we outline a model that enables the utilization of component-based principles even for embedded systems with high optimization demands. The model supports the creation of component variants optimized for different scenarios, through the introduction of an entrance preparation step and an ending verification step into the component design process. These activities are proposed to be supported by tools working on metadata associated with components, where the metadata is possible to automatically retrieve from many development tools.

This paper outlines the theoretical model that is the basis for our current realization work.

Keywords. Optimization, Embedded, Reuse, Component-Based Software Engineering

1. Introduction

Component-Based Software Engineering (CBSE), promises independent development and reuse of software components [7]. The foundation is that general components are reused in many applications, and that problems with architectural mismatches can be eliminated [9]. However, there are studies, e.g., [5, 6, 15] indicating that the development of reusable components in comparison with optimized components for certain applications requires up to five times the effort. A substantial part of the extra effort involves development addressing potential future usage scenarios.

Due to extra-functional requirements present in embedded systems, software must often be optimized and tailored for each application [20]. Embedded systems are often produced in high volumes, implying that smaller memory capsules and cheaper processors has high impact on the total production cost. To enable the verification of other extra-functional properties, e.g., reliability, safety, and timing; design choices must be simple in order to enhance predictability, testability, and analyzability. Thus, reusable components, which are bigger and more complex, are often discarded for optimized solutions.

There are several promising component technologies for embedded systems, e.g., the Rubus component technology [13], Koala [17], and our research prototype SaveCCT [1]. These technologies proves that different important needs for embedded systems can be satisfied, e.g., real-time support, and resource efficient run-time systems. However, in industrial case-studies where SaveCCT have been applied, we have found that much of the necessary support is provided (or possible to provide) but that the need to optimize components for certain applications remains a challenge [1].

The optimization problem has also been recognized in related research, and a classification of different techniques is presented in [11]. Common for many of these techniques is the support for configuration of components, e.g., [2, 3]. However, the flip-side is that future scenarios must be predicted, and that the configuration code increase complexity and thereby resource usage. The other main principle for existing techniques is to apply external adaptation through wrappers [22], or adaptors [21]. The main limitation here is that optimization of the component's internal realization is not possible, e.g., it is not possible to remove functionality. Thus, these techniques it is not suitable for resource constrained embedded systems.

To address the problem, we are creating a framework supporting engineering activities related to optimization and adaptation of components. The framework should be used in combination with a component technology, in our case it will be a part of SaveCCT. In this paper we present the founding model for the framework, and this model is the contribution. The model is based on using component metadata, most of which can be automatically retrieved from development tools. Associating metadata with components is common, e.g., the MS .Net

framework [16] uses metadata for certain run-time properties. In [18] it is showed how metadata can be used to improve the test phase. In our work we use that idea and extend it to cover the whole component development phase. Similar to Built-In-Test (BIT) [4, 8, 19], our model includes reuse of tests, but as specifications and results in the metadata instead of executable test cases embedded in the components. In an initial phase of component design, our model supports preparation activities such as selection of a suitable candidate component to adapt, given a set of requirements forming a new usage scenario. This initial phase provides an estimate of the amount of specialization that must be performed. The need for similar component retrieval support has also been recognized in, e.g., [12], [14]. During component design our model collects key metadata from the tool-suite, in the design, realization, and test phases. At the end of the process the model supports verification activities such as detection of side-effects that have occurred during the specialization process.

In section 2, an overview of the proposed model is given. In section 3, the central distinction of components, variants, and versions is defined. Section 4 presents the metadata that is a core part of the model, while algorithms using the metadata are presented in section 5. Section 6 demonstrates the model by an example. Finally section 7 concludes the paper.

2. Model Overview

Figure 1, shows a schematic overview of the suggested model, fitted into a design process for software components. Characterizing for CBSE is that component development and system development (using components) are separated activities. It is important to be aware of that the focus in this work is on the component development process, and that the majority of the research targeting software components are concerned with system development using components Referring to the figure, the shown design process prior the integration of our model can be imagined as a waterfall model with four steps, design, realization, test execution, and finally delivery to the component repository. The main characteristics to emphasize after the introduction of our model are:

- There is a preparation step added as an entrance step into the process. At this stage, given the requirements forming a new usage scenario, the decision to create a component from scratch or to select a component to specialize are taken through evaluation of the amount of work needed for specialization. The output from this step is a plan, or work-order, guiding design and verification efforts.
- There is an additional verification step at the end of the process. Here unplanned side-effects (not according to the plan from the work-order) are detected, e.g., functionality that has changed without intention in a specialization.
- The model is based on metadata, which is automatically retrieved in the design process. Given that tools are capable of exporting data, the need for manual intervention is small.

Not shown in the figure, but also a central concept, is that the model distinguish components from variants and versions in the repository. This is described in next section.

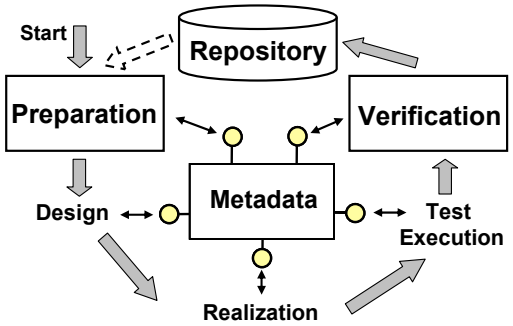

Figure 1, overview of the model

3. Components, Variants, and Versions

In the repository a component may exist in several variants and versions. An overview of the repository is shown in figure 2. The elements shown in the figure are defined below:

- The repository Rep = $\{C_1, \ldots, C_n\}$. The repository on the top-level stores all components in a flat set. The structure is flat since the components have no dependencies to each others in contrast to, e.g., object-oriented approaches were the inheritance relations may affect the storage structure.
- C_i is an abstract component. It is a root node in the repository representing all variants of the i^{th} component in the repository. $C_i = \{C_{i1}, \ldots, C_{in}\}$. The structure is flat indicating no interdependencies between the different variants; they are separate units for usage and maintenance.
- Each variant may exist in several versions $C_{ij} = \{C_{ij1}, \ldots, C_{ijn}\}$. Versioning of the variants is handled according to the rules of common version management theory. The version created latest in time will have the highest version number.
- Referring to a component C_{ijk}, means version k of variant j of the i^{th} component in the repository. C_{ijk} is a concrete component in a component

technology, e.g., [1][17], according to common component definitions, e.g. [10].

Assume that the function Req(x) gives the set of uniquely identified requirements fulfilled by element x. How this is realized is described in the next section. The following guarding conditions must be fulfilled for a software element to qualify as a variant, or version of a component respectively:

Commonality guard - for all variants j and versions k of component i, $\{\cap_{jk} \text{Req}(C_{ijk})\} \neq \varnothing$. This implies that there must be at least one requirement in common between all variants and versions of a certain component. If this guard is not fulfilled, the variants and versions cannot be stored under same component.

Compatibility guard - for a new version k+1 of variant k, $\text{Req}(C_{ijk}) \subseteq \text{Req}(C_{ijk+1})$. This implies that a new version of a variant should fulfill at least the same requirements as the previous version. When this strict guard is fulfilled the new version is backwards compatible with the older version, typically bug-corrections and improvements will sort under this category. In our model, if this guard is not fulfilled the component may be qualified as a new variant; otherwise a new component should be created.

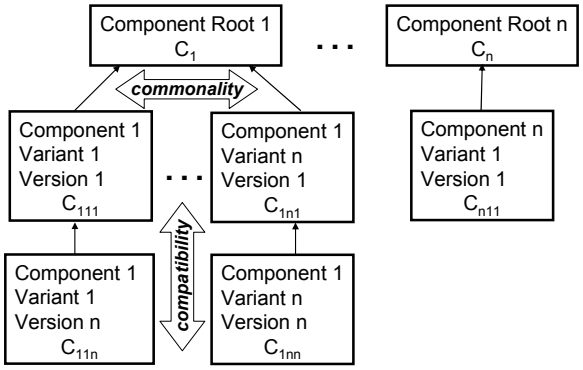

Figure 2, repository layout

4. Metadata definition

Metadata units are associated with all concrete components, i.e., all versions of all variants of a component. The metadata manage requirements, elements of design and verification of the software in the repository.

An overview of the metadata is shown in figure 3. The figure show more metadata compared to what will be formally defined in this paper; this is to give an idea of the overall concept. The core metadata (thicker lines in the figure), are the necessary parts required to provide the support that is emphasized in this paper. Non core parts may be useful when browsing the repository, e.g., containing abstract, keywords, usage statistics, and key design patterns practiced when the component was developed .

To define the core parts of the metadata, let $\mathbf{M_{ijk}} = (\mathbf{S_{ijk}}, \mathbf{G_{ijk}})$ be the metadata associated with C_{ijk}. $\mathbf{S_{ijk}}$ is a specification of the component $\mathbf{S_{ijk}} = (\mathbf{R_{ijk}}, \mathbf{D_{ijk}}, \mathbf{V_{ijk}})$ where:

- $\mathbf{R_{ijk}}$ is a set of uniquely identified requirements $\mathbf{R_{ijk}} = \{r_1, \ldots, r_n\}$. $\mathbf{R_{ijk}}$ contains all documented requirements that the software element tries to fulfill, including both functional and extra-functional requirements. The actual formulation or semantics of the requirement is not strictly required. The important matter is that a unique identity is associated with each requirement.
- $\mathbf{D_{ijk}}$ is a set of uniquely identified architectural entities $\mathbf{D_{ijk}} = \{d_1, \ldots, d_n\}$. Depending on the realization of the software element, these design entities can be different artifacts, e.g., functions, data structures, objects, components or analysis. As for requirements, design entities must be associated with a unique identifier.
- $\mathbf{V_{ijk}}$ is a set of uniquely identified verification cases $\mathbf{V_{ijk}} = \{v_1, \ldots, v_n\}$. $\mathbf{V_{ijk}}$ includes all test-cases, together with expected results, and also obtained results after the test phase. As for $\mathbf{R_{ijk}}$ and $\mathbf{D_{ijk}}$ each case needs to be represented.

$\mathbf{G_{ijk}} = (\mathbf{CR_{ijk}}, \mathbf{VR_{ijk}})$, contains manually defined relations, over the automatically derived sets $\mathbf{D_{ijk}}$, $\mathbf{R_{ijk}}$, and $\mathbf{V_{ijk}}$.

- $\mathbf{CR_{ijk}} \subseteq \mathbf{D_{ijk}} \times \mathbf{R_{ijk}}$ represents the *causal relationships* between elements of the design and their respective requirements. It represents the reason, or the cause, for design elements to exist.
- $\mathbf{VR_{ijk}} \subseteq \mathbf{V_{ijk}} \times (\mathbf{R_{ijk}} \cup \mathbf{D_{ijk}})$ represents the *verify relationships* from elements of the verification cases, to which requirements and/or design entities, each case verifies. Relations from $\mathbf{V_{ijk}}$ to $\mathbf{D_{ijk}}$ represent white-box test cases, while edges from $\mathbf{V_{ijk}}$ to $\mathbf{R_{ijk}}$ represent black-box cases.

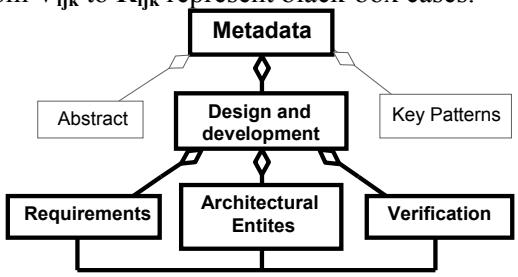

Figure 3, metadata associated with C_{ijk}

5. Central algorithms on the metadata

Figure 1 emphasized the support provided first and last in the component design process, the algorithms applied in the two different stages are described in the following sub-sections.

5.1 Preparation

When a need for a new component is detected, a central decision is to decide if the new component should be obtained through adaptation of an existing component or if a new component should be developed. To support this decision, the metadata can be used to compare candidates for reuse and adaptation. The following expressions determine what requirements that are addressed by variants and versions of a specific component:

- $\mathbf{SR_i}$ derives all requirements shared by all variants and versions of a component. It is defined as the intersection of all requirements addressed by all entities of a certain component C_i: $\mathbf{SR_i} = \{\cap_{jk} Req(C_{ijk})\}$
- $\mathbf{NR_i}$ is the set containing the requirements addressed only by a sub-set of the variants and versions of a certain component C_i. $\mathbf{NR_i} = \{\cup_{jk} Req(C_{ijk})\} - \mathbf{SR_i}$
- **AR(r)** gives the set of versions and variants that address the requirement **r**, of a certain component C_i, $\mathbf{AR(r)} = \{ C_{ijk} \mid \{r\} \subseteq Req(C_{ijk})\}$

The application of the expressions above provides overview information about the components. We can see divide requirements into those addressed by all versions and variants, and those requirements addressed by certain sub-sets. With this information developers are guided in the choice of candidate components to investigate in the work to find a suitable component to reuse.

It is possible to derive a work-order for each concrete component, i.e., C_{ijk}. Initially work-orders are used to estimate the amount of work to apply changes to certain concrete components to fit a new usage scenario. Thus, finding the most feasible candidate to adapt is supported by comparison of work orders. A component whose work order shows little need for adaptation is likely a suitable starting point for a new variant. Later, during the development, the work is guided by the work-order. For a certain concrete candidate C_{ijk}, and given the requirements forming a new usage scenario, the work order show what design entities and what test cases to reuse as-is, to change, and to remove. It also shows what requirements that remains unimplemented and thus will require new development. The functions that are needed to be applied on the metadata are defined here.

An estimation of consequences of a changed requirement, **r**, in terms of the set of affected design entities, **AD(r),** and set of affected test cases, **AT(r),** is determined through:

- $\mathbf{AD(r)} = \{x \mid \mathbf{CR_{ijk}}(x,r)\}$
- $\mathbf{AT(r)} = \{ x \mid \mathbf{VR_{ijk}}(x,r)\}$

The consequences of a removed requirement, **r**, in terms of affected design entities can similarly be determined by the same expressions. However, to determine if the deign entity or test case is not only affected, but according to the relationships expressed in the graphs can be removed, we must take the whole set of all removed requirements into consideration. Let **RR** be the set of requirements that is planned to be removed. Design entities that may be removed are determined through the function **RD(RR)**. Similarly test-cases that may be removed are derived by the function **RT(RR)**.

- $\mathbf{RD(RR)} = \{ x \mid \neg\exists r : [\mathbf{CR_{ijk}}(x,r) \wedge r \in \mathbf{R_{ijk}}\text{-}\mathbf{RR}] \}$
- $\mathbf{RT(RR)} = \{ x \mid \neg\exists r : [\mathbf{VR_{ijk}}(x,r) \wedge r \in \mathbf{R_{ijk}}\text{-}\mathbf{RR}] \}$

5.2 Verification

When a resulting variant or version is created based on reuse of another, it is possible to detect un-planned effects of the changes. To detect unplanned side-effects that may have occurred in the process, regression testing is applied based on information in the work order. The only allowed changes between the results of reused test cases are those we knew would be affected in the work order. If any other changes are detected, they must be investigated. There can be one of two reasons that must be corrected by the developers:

- Unplanned or unnecessary parts were changed during the development of the new variant, which must be found and corrected.
- Undocumented dependencies in the relations $\mathbf{CR_{ijk}}$ and/or $\mathbf{VR_{ijk}}$ should be updated and added to achieve a continuous improvement of the decision supporting relations. It may also be useful to store statistics when undocumented dependencies are discovered, to estimate a precision for work orders.

6. Usage Example

Now that we have defined the elements in the model we will demonstrate the support for design decisions. We do this through a simplified industrial case.

6.1 Initial Component

As a part of an order of a larger system, a component providing an interface to a CAN chip is ordered forming requirements $\mathbf{R_{ijk}}$ as below:
$\mathbf{R_{ijk}} = \{(11, Send),$

(12, Receive),
(13, EnableRemoteReply),
(14, worst case latency for Send 1 ms)}

Given that this component is built from scratch, and stored in an empty repository, i.e., **Rep** = {C_1}, where C_1={C_{111}}. Depending on the developers design decisions, $\mathbf{D_{111}}$, and $\mathbf{V_{111}}$ of the local metadata associated with C_{111} may have the following structure in the repository:

$\mathbf{D_{111}}$ = {(21, FrameTypes),
(22, ReceiveBuffer),
(23, Send),
(24, Receive),
(25, EnableRemoteReply),
(26, Analysis of send, result 500ms)}

$\mathbf{V_{111}}$ = {(31, receive buffer test, *expected*: oldest dropped, *observed*: oldest dropped),
(32, send test, *expected*: all sent *observed*: all sent),
(33, receive test, *expected*: id sequence 2,3,66, *observed*: 2,3,66),
(34, remote reply test, *expected*: remote frame 2, *observed*: remote frame 2),
(35, timing analysis send, *expected*: <500ms, *observed*: 450ms)}

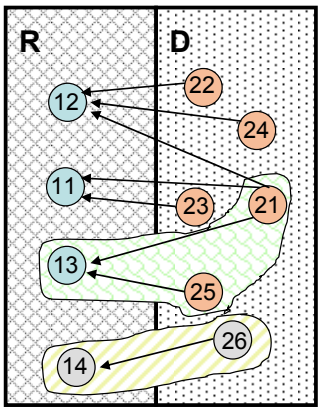

Figure 4, causal relations

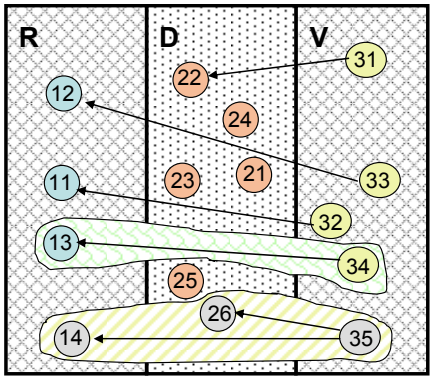

Figure 5, verify relations

Notice that the requirements, design, and verification sets $\mathbf{R_{111}}$, $\mathbf{D_{111}}$, and $\mathbf{V_{111}}$ should be automatically created, given that it is possible to export data from the tools. However, the causal and verification relations $\mathbf{CR_{111}}$ and $\mathbf{VR_{111}}$ remain to be manually defined. These relations can be presented and created graphically, through directed graphs. $\mathbf{CR_{111}}$ and $\mathbf{VR_{111}}$ for the case are defined below, and the corresponding graphs are shown in figure 4 and 5 respectively. For now, ignore the fields surrounding, e.g., nodes with id 14 and 26.

$\mathbf{CR_{111}}$ = {(21,11), (21,12), (21,13), (23,11), (22,12), (24,12), (25,13), (26,14)}

$\mathbf{VR_{111}}$ = {(31,22), (32,11), (33,12), (34,13), (35,26), (35,14)}

The metadata unit for C_{111} is now complete $\mathbf{M_{111}}$ = (S_{111},G_{111}) = ((R_{111}, D_{111}, V_{111}), (CR_{111}, VR_{111})).

We have now the initial version of a variant of a component that can not only be reused. The component is also prepared for adaptation and specialization to form new variants addressing sets of requirements forming other usage scenarios.

6.2 New Component

In negotiation with another customer at a later point in time, the requirements on a similar component as a part of another system forms R_{ijk} as below. Requirement id 13 has changed, indicated here only by a "*", requirement id 14 has been removed, and requirement id 15 has been added.

$\mathbf{R_{ijk}}$ = {(11, Send),
(12, Receive),
(13, EnableRemoteReply*),
(15, GetRemoteFrameStatistics)}

Applying the expressions in section 5.1, the work-order contains the information in table 1. The results from the expressions are visualized in figure 4 and figure 5. The fields in the figures surrounding certain relations show the same as the table, e.g., that due to changes in requirement id 13, design entities {21, 25} may be affected as well as test case 35.

Table 1, A work order for the specialization

	Design Entities	Test Cases
Reuse ids	{22, 23, 24}	{31,32,33}
Affected ids	{21, 25}	{34}
Remove ids	{26}	{35}
Covered requirement ids: {11,12,13}		
Uncovered requirement ids: {15}		

The process may proceed guided by the work order, eventually when tests are complete the results are verified according to section 5.2. In this case according to the work-order it is expected that test case 34 might show other results,

and that observed results of cases {31, 32, 33} should be unchanged.

7. Conclusions

We are convinced that component-based principles are beneficial for all types of software. Mature engineering disciplines always use standardized components. One of the most important prerequisites for component based principles is that components are general, so that they can be (re)used many times. This prerequisite has shown be hard to meet in development of certain software, e.g., embedded software with high specialization demands.

This paper introduces a model that supports developers of embedded software components in using optimized variants of components. The benefits are achieved by introducing a start and a completion step into a regular design flow. The completion-phase provides automatic detection of accidentally introduced side effects in redesign. The starting phase supports the selection of the best matching candidate from a repository of components given a set of requirements.

The model is based on associating metadata with components, and can be highly automated and integrated in an existing development tool-suite, given that it is possible to export data from the tools. An industrial case study is planned, where a prototype realization will be integrated in an existing tool-suite at a sub-contractor company. A sub-contractor company is often faced with challenges in adapting and customizing components to the different needs of customers with varying system architectures and choices in technology and standards. Managing adaptation and optimization of components is therefore a key value for sub-contractors.

References

[1] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, M. Tivoli, The SAVE approach to component-based development of vehicular systems, Journal of Systems and Software, Elsevier, 2006

[2] J. Bosch, Superimposition: A component adaptation technique, Information and Software Technology, 41(5), March 25, 1999

[3] K. Cooper, J. Zhou, H. Ma, I. L. Yen, and Farokh Bastani, Code parameterization for satisfaction of QoS requirements in embedded software, Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, 2003

[4] Component+, http://www.componentplus.org

[5] I. Crnkovic, Component-based Software Engineering - New Challenges in Software Development, Software Focus, John Wiley & Sons, Dec, 2001

[6] I. Crnkovic, M. Larsson, A Case Study: Demands on Component-based Development, Proc. 22nd Int. Conf. on Software Engineering, 2000

[7] I. Crnkovic, M. Larsson, Building Reliable Component-Based Software Systems, Artech House publisher, 2002, ISBN:1-58053-327-2

[8] K. J. Fernandez, V. H. Raja, and M. Morley. A system level testing modeling mechanism in a reengineering environment. Journal of Conceptual Modeling, 2001

[9] D. Garlan, R. Allen, J. Ockerbloom, Architectural Mismatch or Why it's Hard to Build Systems Out of Existing Parts, Proc. of the 7th Int. Conf. on Software Engineering, Apr, 1995

[10] G. T. Heineman, W. T. Councill, Component-based Software Engineering, Putting the Pieces Together, Prentice-Hall, 2001, ISBN: 0-201-70485-4

[11] G. T. Heineman, An evaluation of component adaptation techniques, 2nd ICSE Workshop on Component-Based Software Engineering, 1999

[12] Y. Li; J. Yin; J. Dong, A Component Management System for Mass Customization, 1st Int. Symp. on Computer and Computational Sciences, April, 2006

[13] K.L. Lundbäck, J. Lundbäck, M. Lindberg, Component-Based Development of Dependable Real-Time Applications, Arcticus Systems: http://www.arcticus.se

[14] H. Mili, F. Mili, A. Mili, Reusing software: issues and research directions, IEEE Trans. on Software Engineering, Jun, 1995, 21(6)

[15] M. Mrva, Reuse Factors in Embedded Systems Design. Siemens AG, Munich, Germany, 1997

[16] MSDN homepage, http://www.msdn.com

[17] R. van Ommering, F. van der Linden, J. Kramer, The Koala component model for consumer electronics, IEEE Computer, March, 2000

[18] A. Orso, M. J. Harrold, D, Rosenblum, G. Rothermel, M. L. Soffa, H. Do, Using Component Metacontents to Support the Regression Testing of Component-Based Software, Proc. IEEE Int. Conf. on Software Maintenance, Nov, 2001

[19] Y.Wang, G. King, D. Patel, S. Patel, and A. Dorling. On coping with real-time software dynamic inconsistency by built-in tests. Annals of Software Engineering, 7(1), Oxford, 1999.

[20] W. Wolf, What is embedded computing, IEEE Computer, 35(1), Jan, 2002

[21] D. M. Yellin and R. E. Strom. Protocol Specification and Component Adaptors. ACM Trans. on Programming Languages and Systems, 19(2):292-333, March 1997

[22] J. Brant, B. Foote, R. e. Johnson, D. Roberts, Wrappers to the Rescue, Proc. 12th European Conf. Object-Oriented Programming ECOOP'98, July, 1998