Technologies for Evolvable Software Products: The Conflict between Customizations and Evolution

Peter Sestoft and Sebastien Vaucouleur

IT University of Copenhagen, Denmark
{sestoft,vaucouleur}@itu.dk

Abstract. A *software product* is software that is built for nobody in particular but is sold multiple times. A software product is typically highly customizable, or adaptable, to particular use contexts; moreover, such a software product can typically be thought of as a common *kernel* plus a number of *customizations*, one for each use context. A successful software product will be used for many years, and hence the kernel must evolve to accommodate changing demands and environments. The subject of this paper is the conflict between the *customizations* made for each use context and the *evolution* of the kernel over time. As a case study we consider Microsoft Dynamics AX and Dynamics NAV, highly customizable enterprise resource planning (ERP) software systems, for which upgrades are traditionally costly. We study the challenges related to the customization/evolution conflict and present some software engineering approaches, programming language constructs and software tools that attempt to address these problems, and discuss whether they could be brought to bear on the conflict.

1 Introduction and Definitions

A successful software product is typically released in many versions over many years; it *evolves* over time. Also, a software product is typically *customized* to permit effective use in many different applications and contexts. In this work, we are interested in the problems and conflicts that arise from the combination of evolution and customization; we call this the *upgrade problem*.

In this section, we define the most important terms used in the next sections. Then we discuss the relation to the concept of a software product line as it is currently used in the literature. Finally, we outline the contributions and the structure of the paper.

Definitions. A *software product* is software that can be used in many different contexts, such as a shared calendar system for organizations, or a text processing system. Such software products should be contrasted with software that has been developed in a project for a particular purpose, for instance for the securities trading desk of a particular bank. One may view a particular instance of a software product, deployed in a particular context or organization, as consisting

E. Börger and A. Cisternino (Eds.): Software Engineering, LNCS 5316, pp. 216–253, 2008.

of a software *kernel*, plus *customizations* (adaptations of the kernel to the context), plus possibly further *configurations*, whether organization-wide or for the individual end-user. In this paper we shall distinguish *customization*, which can add new and possibly unforeseen features to software, from *configuration*, which enables or disables features that are already present in the software, change their behaviour, or affect the way they appear to the end-user[1]. *Software evolution* is the phenomenon that software must change over time to stay useful: errors must be fixed, security holes must be plugged, new functionality must be supported, and changes in the environment must be accommodated [25]. Finally, software composition is the construction of software applications from existing software parts.

Software product lines. The software products considered in this paper are clearly related to software product lines [17]. Software product lines typically have a closed-world assumption in which a central agent (such as a chief engineer) has a clear idea of all the variations that are required. We will consider this approach to customization and compare it with approaches that have an open-world assumption – that is, where no central agent has a clear understanding of all the possible customizations that may be needed for a software product.

For software product lines with a closed-world assumption, the construction of a particular member of the product line comes down to choosing from a predefined set of features, that is, to configuration. In that case, kernel evolution and customization are hard to distinguish.

Contributions. The contribution of this paper is two-fold. First, it gives a more precise characterization of what we call the upgrade problem. Domain experts and practitioners have previously claimed that the upgrade problem is an important one, but surprisingly it has never been thoroughly studied from a technical angle to the best of our knowledge. Second, we give a subjective evaluation of some of the most commonly used customization techniques and study how they can be used to mitigate the upgrade problem. We support our conclusions by using an explicit set of criteria, as well as a simple running example.

Roadmap. The next section gives a detailed explanation of the upgrade problem. Then, section 3 gives a concrete example of this problem, through a study of two widely used ERP systems. Section 4 gives a list of criteria that will be used in section 5, the core of the paper, to give a subjective evaluation of some of the most widely used customization technologies.

2 The Upgrade Problem

The focus of this work is the interaction of two distinct dimensions of change, namely *customization* and *evolution*. When the kernel of a software product

[1] Note that this terminology is not universally accepted. For instance, the Microsoft Excel 2003 menu `Tools > Customize` performs what we would call configuration: it determines which toolbars to display in the user interface, and so on.

evolves, and an organization wants to upgrade to a new version of this kernel, the customizations of the deployed software product must be carried over to the new version. In simple cases this may just involve copying the customizations over unchanged, but in general it may involve a rewrite of the customizations and also require comprehensive knowledge of the customizations as well as the old kernel and the new kernel; see section 3.9. This work incurs considerable cost and often causes end-user companies to postpone the upgrade as long as possible.

2.1 Customizable Software

Almost all software is *configurable*. Even the most mundane of applications, the Minesweeper game delivered with Microsoft Windows, has several levels of difficulty, sounds on or off, and so on. Also, there is hardly a Unix (or Linux) program without a configuration file somewhere in `/etc/`, or a `.foorc` configuration file in the user's home directory.

Moreover, much software is *customizable*, in the sense that it admits subsequent extensions of its functionality, unforeseen at the time the software itself was designed, implemented and shipped. For instance, Web browsers such as Firefox support *add-ons* that enable the browser to display new media types; spreadsheet programs such as Microsoft Excel support *add-ins* that enable the spreadsheet program to solve optimization problems and other specialized tasks; and integrated development environments such as Eclipse support *plug-ins* that enable the development environment to support new programming languages, graphical modelling tools, and so on. These add-ons, add-ins and plug-ins are what we call customizations.

In all the above examples the additional functionality is provided via software components that can be *dynamically* loaded into a running application on demand. A more *static* approach would permit customization when the software is built, by importing (or not) third party features into the software when it is compiled or linked. Operating systems such as Linux support both static and dynamic customization: drivers for particular network devices, file systems, and so on can be added to the Linux kernel when compiling it, and in addition some modules (e.g., wireless network drivers, support for USB devices) can be loaded and unloaded on demand while the operating system is running.

In the above examples, there are well-specified interfaces between the kernel and the software (add-ins, add-ons, plug-ins, modules) that implement the additional functionality. For some software systems it is difficult or impossible to foresee what kinds of customizations are needed, so it is impossible to design interfaces that are both general enough and specific enough. Instead, (some) customizations require edits to the kernel software itself. We shall call the latter a *white-box approach* to customization of the kernel.

2.2 Software Evolution

All software will change from time to time, if it is used at all, as evidenced by the all too familiar and increasingly arcane version numbers: C# compiler

version 3.5.21022.8, Eclipse version 3.3.1.1, Oracle database version 9.2.0.6.0, Linux kernel 2.6.23.1-42.fc8, and so on.

Software evolution is a research topic in its own right, pioneered by Lehman in an empirical research setting three decades ago [24,23], and now having its own conferences, terminology and methods, as well as a *Journal of Software Maintenance and Evolution*. Lehman's original software evolution research made several observations: We all too often believe that the system we are currently building will be the final one, and hence we fail to plan for change, whether foreseeable or unforeseeable. Also, the very purpose of some kinds of software systems, called E-programs by Lehman [24], is to cause a change in the context in which they are deployed, and hence those are even more prone to change, as the context changes and feeds back change requirements on the software system. Current research on software evolution and maintenance attempts to classify the various reasons for evolution [25], to propose theoretical means to understand software evolution and to find practical mechanisms to help maintain software during evolution.

Probably the strongest drivers of software evolution are:

- commercial pressure to support additional functionality
- organizational changes, such as company mergers
- legal changes, such as additional audit requirements
- changing technical environments, such as evolving operating systems
- demand for distributed and mobile access and new user interface technology
- co-evolution for interoperability with other software

2.3 The Evolution of Specifications

The problem of supporting customization as well as evolution cannot be addressed without taking evolving *specifications* into account. In an ideal software architecture, every software kernel component is accessed only through a well-defined specified interface. If a customization modifies a component, but the component continues to satisfy the specified interface, then obviously the software system continues to work correctly, using the traditional relative notion of correctness (satisfaction of a specification).

However, the point of software evolution is often that *the specification* must change, not just the implementation. Changes to the specification are usually caused by changes in the environment, such as new business processes or user needs, as outlined in section 2.2. When the specification changes, the black-boxing of the implementation behind the specification provides little help in the upgrade of customizations.

The more interesting and challenging case is when the software kernel evolves due to a changing specification, not the case where its implementation changes but the interface specification remains the same.

2.4 Upgrade Problems in Operating Systems

In early versions of Microsoft Windows, upgrade problems would be experienced almost daily, a phenomenon that was known under the name "DLL hell". Most

applications would rely on dynamically loaded libraries (DLLs), which were typically shared system-wide between multiple applications. This caused problems because at any time there could be only one installed version of each DLL, and newer versions of a DLL were not necessarily backward compatible. For instance, installing a new version of the Internet Explorer web browser might require an upgrade also of a DLL, and the deletion of the old version. Subsequently one would discover that the accounting software installed on the same computer had relied on that old version and was incompatible with the new version, and hence stopped working. At its core, the problem was that multiple applications relied on a common resource (DLL), and that one application would affect the others through unwanted modification of the common resource. Another variant of this problem would be that manipulation of the PATH environment variable caused by installation or upgrade of one application would mean that other applications could no longer locate their DLLs and therefore stopped working.

The same problems could be observed in early Linux distributions, where an upgrade of the `gcc` C compiler and its associated libraries might break some other part of the system. In more recent versions of Microsoft Windows as well as Linux, such problems are addressed by allowing multiple versions of the same library to coexist. For instance, in a current Linux installation one may find both versions 0.9.7a and 0.9.7f of the `libssl.so` library.

Modern programming platforms, such as Microsoft's .NET, address these problems in an even more powerful way, by allowing one library (called an assembly) to express its versioned dependencies on other libraries. A forthcoming version of the Java platform is expected to support versioning of libraries (called modules) and versioned dependencies in a similar way [15]. In the experimental language Fortress being developed at Sun Microsystems for DARPA, the basic program module is the trait (see section 5.6), and the language aims to provide upgradable program components in the form of versioned collections of traits [2,3].

2.5 Conclusion on the Upgrade Problem

The upgrade problem is found in many contexts and can be addressed in many ways. In the remainder of this paper we focus on software products, and in particular on the conflict between customization of a software kernel and subsequent evolution of that kernel. In particular, we consider this problem in relation to highly customizable enterprise software systems.

3 Case Study: Dynamics AX and NAV

To get a more concrete setting for discussing upgrade problems, we now present Microsoft Dynamics AX [26] and Dynamics NAV [27,38], two enterprise resource planning (ERP) systems from Microsoft Corporation.

For short, the term "Dynamics" will refer to the Dynamics products (AX and/or NAV), and the term "Dynamics developers" will refer to the core Dynamics development teams at Microsoft.

3.1 Add-ons and Customizations

Both Dynamics AX and Dynamics NAV are highly customizable and configurable, and customization takes place in several stages. Microsoft builds and sells a kernel system, consisting of runtime environment, database system, development environment and a number of core packages, e.g., for sales tax reporting in a particular country. A large number of partners, also called independent solution vendors (ISVs) or value-added resellers (VARs), sell add-on solutions and customizations.

An *add-on solution* may be targeted to a particular industry (a vertical solution area), such as apparel and textiles, or address a particular activity within an organization (a horizontal solution area), such as customer relationship management. Several add-on solutions may be used together in a Dynamics installation. Simply put, in ERP parlance an add-on is a set of customizations.

Further *customizations* may be created on top of the kernel and the add-ons, thus tailoring the ERP system to the needs and processes of a particular company. Some end-user companies even make such customizations themselves.

Add-ons are written as additional modules or by modifying parts of the kernel modules, using the development environments. Hence, Dynamics AX and NAV are software products developed over a long time and sold in many copies, with a wide range of customizations, to many different customers. They also exhibit the upgrade problem outlined in section 2 above, in a particular way: The add-ons and customizations are developed primarily by partner companies, whereas the kernel evolution is controlled primarily by the Dynamics development team.

This section will provide details about the Dynamics software products, the upgrade problems experienced, and some current practices to alleviate them.

3.2 Dynamics NAV Versus Dynamics AX

Before we dive into the upgrade problems in more detail, let us consider the characteristics of the Dynamics NAV and Dynamics AX enterprise resource planning systems. Both systems are partially model-driven and partially programming language based. Namely, database tables, runtime data structures, and the user interface (forms) are described by metadata, not built using programming language declarations. On the other hand, behaviour is described using traditional programming language constructs, called *code units*, which correspond to functions or methods.

The two systems have distinct organizational and technical characteristics:

- Dynamics NAV mostly targets smaller organizations, for which pre-developed add-ons mostly suffice, so they only require minor customizations. A large number of organizations run Dynamics NAV. The integrated development environment is called C/SIDE, and the programming language, C/AL, is a relatively simple language with a Pascal-like syntax. The developers employed by NAV partners usually focus on the customer's business and many do not have a strong background in software development. Code unit customizations are made simply by editing the required code units in the C/AL language.

– Dynamics AX mostly targets larger and more complex organizations, that often require extensive customizations. Fewer organizations use Dynamics AX than NAV. The integrated development environment is called MorphX, and its proprietary programming language X++ is an object-oriented language with a Java-like syntax. The developers employed by AX partners often have a good background in software development. The Dynamics AX model is structured into a number of layers, with layers for the kernel, layers for partners' customizations, layers for further customizations in the end-user organization, and so on; see section 3.8. A code unit customization is made by copying the code unit from the layer at which it was originally defined and then adding and editing at a higher layer. The higher layer version will then be used instead, and is said to shadow the lower layer code unit; see section 3.9.

We present both systems here, because their different organizational and technical characteristics cause different kinds of upgrade problems.

3.3 The Dynamics Ecosystem

Microsoft and its partner companies form an ecosystem in which the partners depend on Dynamics developers for providing a kernel that is robust, comprehensive, easily customizable, and up to date. Conversely, Microsoft depends on the partners for marketing its kernel, for developing add-ons that make it valuable for customers, for making customizations, and for deploying the customized solutions in customer organizations.

There is a delicate balance in relation to the evolution of the system kernel: If the kernel changes by frequent small steps, then the partners will find it difficult to sell all these upgrades (of kernel and customizations) to their customers; but if the kernel changes by infrequent radical steps, partners or customers may find upgrade so complex that they can just as well switch to a competing product (such as SAP, an Oracle-based system, or software as a service). Also, if the kernel evolves too slowly or not at all, advanced customers may find that it no longer interoperates well with other software they use, or does not support new reporting standards or functionality that they need, such as visualization, business intelligence, electronic trade, etc.

3.4 What Constitutes an Upgrade

Common to Dynamics AX and NAV is that an upgrade to an installation involves upgrade of kernel and customizations as well as conversion of the end-user organization's production data. The data conversion poses interesting challenges itself. First, it is highly time-critical because the end-user company usually cannot conduct business while the data conversion is being done, so the conversion must take place over a weekend or an extended weekend. Second, the data conversion must be fully reliable, or it would disrupt the business. Third, full-scale testing of the scripts that perform the data conversion cannot be conducted until a test environment consisting of the entire upgraded ERP system (new kernel

and upgraded customizations) is available, which is usually late in the process; see also section 3.9. The code and metadata migration can be done in advance of the actual data conversion upgrade; only the data conversion is time-critical in this sense.

Nevertheless, we shall say no more about the data conversion process in this paper, but focus on the problems caused by upgrade of code customizations.

3.5 Upgrade Problems in Dynamics NAV and Dynamic AX

It is clear from a survey of partners [10], from talking to the Dynamics AX and NAV core development teams, and from various online forums and blogs, that upgrade of customizations in Dynamic AX and NAV are problematic. For instance, a public video from a Dynamics AX core developer [34] acknowledges that upgrade of customizations can be costly: "Our research shows that an average upgrade costs as much as 30% of (the original cost of) the customizations". As further evidence, a Google search for `dynamics nav upgrade` gives 114,000 hits (January 2008). There are companies, such as Liberty Grove Software in Illinois, USA, that specialize in doing NAV upgrades for other partners at a fixed price quoted after a preliminary upgrade diagnostic. Also, partner-oriented materials from Microsoft itself suggest that care is needed when customizing the systems to minimize future upgrade problems (see section 3.10).

3.6 Constraints on a Solution to the Dynamics Upgrade Problem

Although a kernel upgrade affects both add-on solutions and partner-made customizations (see section 3.1), in this paper we focus on the problems caused by partner-made customizations, because fewer resources are available for upgrading those than for upgrading add-ons, which are usually sold more than once.

A potential solution to the upgrade problem should work with the current ecosystem (see section 3.3), and should provide a plausible upgrade path from the technologies currently used (the existing code base is very large, therefore incremental technology adoption is important). Ideally the solution, especially for NAV, should support the short edit-compile-run cycle that developers are used to. Developers add, modify and experiment with customizations in the development environment, and then immediately switch back to the running enterprise application without a lengthy build phase and without restarting the enterprise application and loading data anew.

3.7 Handling Upgrade in Dynamics NAV

Here we consider how the modest size and complexity of some NAV customizations mean that the upgrade of customizations can be handled by rather simple techniques. A particular Dynamics NAV partner, Logos Consult in Denmark, reports [28] that most of their original customization projects are small, on the order of 50–500 man hours, and involve only one or two developers. While doing

the original customization, developers simply mark each change in the customized code using stylized change comments with date and developer's initials, like this:

```
// >> 07.FM
DtldCVLedgEntryBuf."Document Date" := "Document Date";
DtldCVLedgEntryBuf."Job No." := "Job No.";
// <<
```

These stylized comments are easy to search for in the source base, and indicate *who* made the change and *when*. Because customization projects are so small, and because developers stay long with Logos Consult, this information is enough for the developer to understand how to upgrade the customization when subsequently the kernel gets upgraded; no special tools are used to assist in the upgrade. Program comments might also be used to indicate *why* the change is made, but often this is not needed.

The Dynamics NAV approach sketched above is simple and suffices for NAV applications that do not differ too radically from the NAV kernel. However, it is unlikely to scale to applications that require extensive customizations, such as customizations that require a large number of places in the kernel source code to be updated correctly.

In the rest of this paper we will focus on Dynamics AX, whose customizations are usually much more elaborate than those for NAV.

3.8 The Layered Structure of a Dynamics AX Application

The Dynamics AX layering system supports multi-stage customization and extension. The architecture has eight layers [14, page 15], shown in Figure 1. An application element (also called model element) at a higher layer hides one with the same name on lower layers. This supports multi-stage customization because a lower-layer application element may be customized at a higher layer, and that customized application element may be further customized at a yet higher layer.

For each of the eight layers shown in Figure 1 there is a patch layer directly above it, used for small delta updates, for instance to avoid redistributing a slightly changed version of the entire 472 MB SYS layer file.

3.9 Customization Using AX Layers

To customize or extend an application element from a lower level (say SYS) at a higher level (say LOS), the developer copies the entire application element to the LOS level and makes the desired edits to it there. Henceforth the system will use that customized application element. A subsequent upgrade to the application element at the SYS level is not automatically carried through, but must be handled manually in an upgrade project.

In response to a subsequent kernel upgrade, at least the following tasks must be performed:

– Find all those lower layer elements that have changed in the new kernel version *and* have been customized in the current installation.

Layer name	Meaning and purpose
USR	User: Individual companies, or companies within an enterprise, can use this layer to make customizations unique to customer installations.
CUS	Customer: Companies and business partners can modify their installations and add the generic company-specific modifications to this layer. The layer is included to support in-house development without jeopardizing modifications made by the business partner.
VAR	Value-added reseller: Business partners use this layer, which has no business restrictions, to add any development done for their customers.
BUS	Business solution: Business partners develop and distribute vertical and horizontal solutions to other partners and customers. A vertical solution targets a particular line of business such as brake pad manufacturing. A horizontal solution addresses a particular task that is similar across multiple businesses, such as car fleet management.
LOS	Local solution: For strategic local solutions developed in-house.
DIS	Distributor: For critical hotfixes.
GLS	Global solution: For country-specific functionality.
SYS	System: The lowest application element layer and the location of the standard Dynamics AX application.

Fig. 1. The layers of a Dynamics AX application. The LOS, DIS and GLS layers are developed by the Dynamics development team but their application elements can be customized by partners. Only Dynamics developers have access to the element definitions at the SYS layer.

- In each case, decide whether
  (a) the new lower layer functionality makes the customization unnecessary; if so, remove it
  (b) the customization continues to work; if so, copy it to a new customization of the lower layer code
  (c) the customization no longer works; if so, design and implement a new one

These steps require insight into both the old and the new version of the Dynamics AX kernel, into the old customizations, and into the reason for making those customizations in the first place. Hence this work must be done by an expert, preferably the same developer who made the old customizations.

A *shadow* is an application element from the standard application that has been modified at a higher level. The cost of an upgrade (of the standard application, say from AX 3.0 to 4.0) is to a high degree determined by the number of shadows [14, pages 464-467].

A partner-oriented textbook on Dynamics AX distinguishes the various environments in which a version of the system may execute [14, page 466]: production environment, test environment and development environment. It also distinguishes the following phases of the upgrade process, from Dynamics AX 3.0 to AX 4.0, say:

1. Test AX 3.0 layer files (customizations) in test environment
2. Create a production environment with AX 3.0 and the layer files

3. Modify layer files to work in AX 4.0; [that is, upgrade the customizations]
4. Write data migration code and migrate data from AX 3.0 production environment to AX 4.0 development environment
5. Perform functional test of the AX 4.0 application with migrated data
6. Move AX 4.0 layer files to production environment and migrate up-to-date AX 3.0 data files; this is the time-critical step mentioned in section 3.4
7. Start production on the AX 4.0 application

3.10 Mitigating Code Upgrade Problems in Dynamics AX

A public video called "Smart Updates" from a Dynamics AX core developer [34] gives some advice on upgrade in Dynamics AX. Its main messages are:

- One should customize small application elements such as class methods, and avoid big ones such as forms: "Once you customize an application element, a copy of the entire original element is placed in the customization layer". The larger application elements one customizes, the more future upgrade liabilities are incurred.
- One should avoid gratuitous customization: "It is tempting to customize everything" but then later the "customer upgrades the kernel application" and "you'll have to resolve all conflicts" that is, "whenever you're overlayering an element that has changed"
- One should avoid, whenever possible, code unit customizations that could cause a conflict at a later upgrade. Instead one should use "class substitution".

"Class substitution" simply exploits that the Dynamics AX language has object-oriented features, unlike the Dynamics NAV language. The idea is to (1) make a derived class of the to-be-customized lower layer base class, overriding the method that should be customized; (2) to introduce a factory method, for instance called "`Construct()`" that returns an object of the derived class instead of the base class object; and (3) to make sure this `Construct` method is called everywhere the base class constructor would otherwise be used. Section 5.1 below further explores this approach to customization, which is a classic object-oriented idea. The point is that a customization based on "class substitution" is much easier to upgrade than a customization that consists of arbitrary edits to the source code of a code unit.

4 Evaluation Criteria

This section describes the four central criteria that we will use in section 5 to evaluate a range of customization technologies.

- Need to Anticipate Customizations (A kernel developer concern.)
- Control over Customizations (A kernel developer and partner concern.)
- Resilience to Kernel Evolution (An end-user concern.)
- Support for Multiple Customizations (A partner and end-user concern.)

Table 1 on page 250 summarizes the evaluation results.

4.1 Need to Anticipate Customizations

Many software engineering techniques for software customization are based on some degree of *anticipation* of future changes. When the designer can foresee some future needs for customization and evolution of the software system, he will choose a software design that can accommodate these with as few changes as possible. Unfortunately, it is not always possible for the designer to foresee well enough the broad class of possible future customizations. In general, there is a trade-off between control and flexibility. For instance, a customization technique that permits arbitrary source code edits offers little control but high flexibility. Conversely, a customization technique that permits only a choice between a number of predetermined options offer high control but little flexibility.

We distinguish approaches that:

- **Require no anticipation.** The customization technique does not require anticipation of the customizations, whether of the customization points nor of the customization kinds.
- **Require anticipation of the customization points.** The customization technique requires the anticipation of the customization points – that is where customizations can be applied in the source code.
- **Require anticipation of the kind of customizations.** In this case, the customization technique expects the developer to foresee the content of the customizations that will be potentially applied.

4.2 Control over Customizations

When a developer is customizing a correctly functioning software system, he takes the risk that his changes break the coherence and correctness of the current implementation. Hence, a customization technique should help in preserving the intent of the original software maker. The customization techniques typically offers control over customization at two different staging times: design-time and run-time. We will categorize the customization techniques according to the following categories:

- **Design time control over the customizations.** Customizations can be constrained during the design stage of the software product's kernel.
- **Run-time control over the customizations.** The customization technique gives explicit support for controlling customizations at run-time (for example activation and deactivation of certain customizations).
- **No control over the customizations.** The technique provides no explicit support for controlling the customizations.

4.3 Resilience to Kernel Evolution

A software product that has been customized will eventually need to be upgraded to a more recent version. Since the kernel will have evolved, it is likely that the customizations cannot be ported automatically to the new version. Different customization techniques have different weaknesses in this respect and require

different amounts of intervention from the developer to port customizations to the new kernel. The third criterion is the resilience of customizations to the evolution of the kernel. We will differentiate the following three categories of explicit support for resilience to evolution of the kernel:

- **Some resilience to evolution.** The customization technique provides some resilience even to evolution of parts of the kernel related to existing customizations.
- **Restricted resilience to evolution.** Resilience only to evolution of parts of the kernel unrelated to existing customizations. Existing customizations may rely indirectly on some part of the kernel that has changed, which may affect the behaviour of those customizations. In some cases this will be intended—after all, the point of changing the kernel is to change the system's behaviour—but in some cases it will be unintended. We assume here that it is impossible to distinguish those two cases by automatic means.
- **No support for resilience to evolution.** The customization technique provides no explicit support for resilience to evolution of any parts of the kernel. Any part of the kernel may have been altered by some customization, so any change to the kernel may conflict with somebody's customization. Inspection (manual or tool-supported) is needed for each customization to detect whether it conflicts with a change to the kernel.

4.4 Support for Multiple Customizations

Very often customizations are not made by the same company. The challenge is that those multiple customizations must be gathered together into a single product. We will distinguish three categories of techniques with respect to support for multiple customizations. First, those who support parallel development (customizations can be independently developed and brought together at a later stage, possibly by an other company). Those who support only sequential development: customization are conceived one after the other. Finally we distinguish the techniques that provide no explicit support for multiple development. We summarize those three categories:

- **Support for parallel development of customizations.** Multiple customizations can be independently developed and then subsequently applied to the same customization point in the kernel. There is still a risk that the customizations have unintended interference, for instance by updating some data structure in the kernel.
- **Support for sequential development of customizations.** If one customization is made after, and has access to the other one, then both can be applied to the same customization point in the kernel.
- **No support for multiple customizations.** No support for multiple customizations without breaking the abstractions that are used for the customizations.

4.5 Runtime Performance Penalty

Runtime performance can be an important criterion, especially for computation intensive software systems and for core software such as collection libraries. However, all the customization technologies considered in this paper have acceptable runtime performance overhead, typically comparable to a few indirections or a virtual method call per customization point reached during execution. This should be contrasted with reflective method calls, which are typically one or two orders of magnitude slower.

Since all the technologies considered here have satisfactory performance, we will not discuss this criterion further.

4.6 Illustration of the Criteria

We describe further the last three criteria through an illustration, see figure 2.

– Figure 2(a) illustrates our second criterion: a software product P_1 is being customized by a third-party programmer and is further customized by

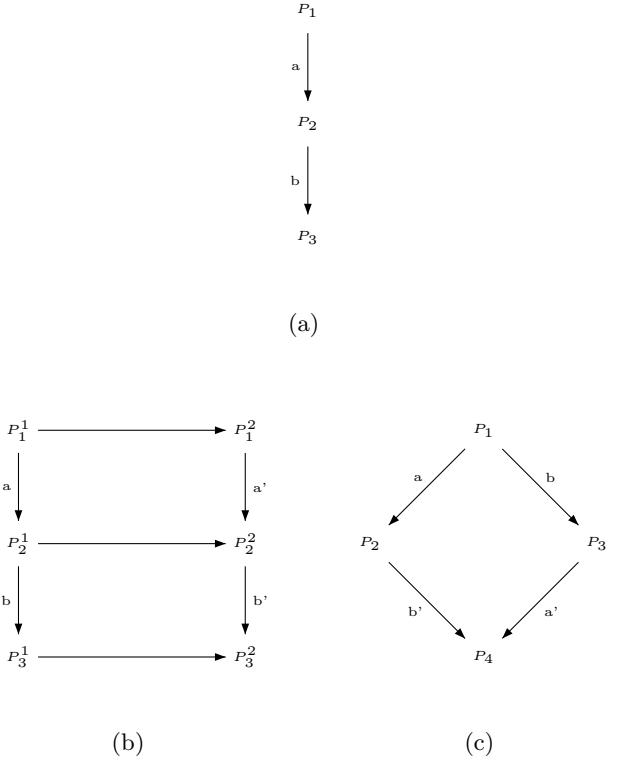

Fig. 2. Concerns (a) Further customization (b) Resilience to Kernel Evolution (c) Support for Multiple Customizations

another programmer, resulting in a software product P_3. The concern here is the staging time of the control for customization: design-time, runtime, etc.

- Figure 2(b) illustrates our third criterion: again, a and b are two successive customizations of an original software product P_1. The original kernel P_1^1 will eventually evolve into a new version P_1^2. The concern here is the ease with which customizations can be ported to the evolved kernel.
- Figure 2(c) illustrates our fourth criterion: here a and b are independently conceived customizations of an original software product P_1. Those two customizations are then used by another company to compose the software product P_4. Informally, the concern here is that the two customizations can be developed independently and brought together at a later stage, ideally yielding an equivalent software product whether one applies a then b', or b then a'. Note that this equivalence is a design goal, not a theorem—to prove such a thing would require a clear definition of the notion of equivalence.

5 Survey of Software Customization Methods

Software customization is a recurrent theme within the software engineering community. Software extension in particular has received much attention from the researchers working on software reuse. Software reuse is important for economical reasons: instead of developing software from scratch one hopes to save effort and obtain better quality by reusing an existing software module, or sometimes an entire software system. They are many different ways to implement customizations. In this section, we review some of these customizations techniques, and we categorize them with respect to the criteria defined in the previous section.

5.1 Inheritance

Inheritance and dynamic binding are heavily used within object-oriented programming to create families of software systems. Virtual methods allow for customization by subclassing. This is essentially the "class substitution" approach for Dynamics AX customization described in section 3.10.

For example, assume we need an `Invoice` class with a `GrandTotal` method that is customizable in the sense that the computed grand total may be modified by a customization. Then we can define a base class `Invoice` with a virtual method `After`, like this:

```
public class Invoice {
  protected virtual void After(ref double result) { /* do nothing */ }

  public double GrandTotal(int input) {
    double total = ...;
    After(ref total);
    return total;
  }
}
```

If we want to customize `Invoice` to give a 5 percent discount on grand totals over 10,000 Euros, we declare a subclass in which `After` has been overridden to do just that:

```
private class CustomizedInvoice : Invoice {
  protected override void After(ref double result) {
    if (result >= 10000)
      result *= 0.95;
  }
}
```

Basically, as is usual in object-oriented programming, the `After` virtual method is a parameter (of function type) of the `Invoice` class, and that parameter may be (re)bound in subclasses. This particular example is a variant of the well-known Template Method design pattern [13].

To ensure that all clients use this customization of `Invoice` one can require them to obtain `Invoice` instances only through a central factory method, using the Factory design pattern [13]:

```
public static Invoice Construct() {
  return new CustomizedInvoice();
}
```

Then only one place in the code needs to be changed when a new customization is created. As a further precaution against clients creating un-customized `Invoice` instances, one could declare the `Invoice` base class abstract.

Hence, customization of methods can be done by method redefinition. Dynamic binding allows for run-time selection of the method body to be executed depending on the actual type of the target object. Multiple dispatch systems such as CLOS claim to be more flexible in that they allow for the selection of the methods upon the types of all of their arguments.

- *Need to Anticipate Customizations.* This technique requires anticipation of the needed customization points. In the `Invoice` example, as in any use of the Template Method pattern, the abstract template method is basically a (function-type) parameter of the class, and one needs foresight to determine which template methods are needed and where they need to be called. Also, the designer of the software system must foresee that the Factory pattern might be required to create an instance of a specific implementation of the `Invoice` class.
- *Control over Customizations.* Correctness in statically-typed object-oriented languages is mainly supported by the type system. The compiler will enforce at design-time that the method to be called exists (no "Method not found" exception at run-time) and that the formal and actual parameters are type-compatible. Hence the control is done at design-time. Other languages (such as Spec#, JML, etc.) allow for behavioral specification by the use of contracts. Contracts are assertions that can be be checked at run-time, or, in some specific cases, verified at compile-time. As an example, we could add

a post-condition to the `After` virtual method to ensure that the customized variant of `Invoice` returns a non-negative value.

```
public abstract class Invoice {
  protected abstract void After(ref double result)
  ensures result >= 0;
  ...
}
```

- *Resilience to Kernel Evolution.* When the abstract class `Invoice` evolves, customized versions of the software system might stop functioning correctly or not even compile any longer. For example, using C#, if the type of the formal parameter `result` in the abstract method `After` in class `Invoice` is changed from `double` to `int`, the compiler will reject the existing customized versions. The current version of C# does not allow any form of variance in the redefinition of formal parameters in subclasses. Now consider the case that the signature of the abstract method `After` does not change in the new version of that base class, but that its post-condition now requires that the result is positive. We say that the postcondition of the abstract base method was strengthened in its new version. Existing customized version of the `Invoice` class that assign zero to `result` now fail to satisfy the post-condition specified in the abstract method. This is likely to only be discovered at runtime, typically resulting in an exception. One may argue that this is the only acceptable output in such a case.
- *Support for Multiple Customizations.* Single inheritance here restricts the customizations to sequential development. More complex design patterns are required to support the composition of independently developed customizations of `Invoice`. The decorator design pattern for example will allow for more flexibility than does inheritance, allowing responsibilities to be added and removed at runtime [13]. Also, a variant of the proxy pattern allows to chain proxies, which provides support for multiple successive customizations. Note that the order in which proxies execute can be crucial for correctness.

The chief *advantage* of the virtual method approach to customization is that it is well understood and supported by mainstream programming languages such as Java and C#. Evolution of the base class does not require any changes to the customizations (subclasses) so long as no base class customization points are removed and no customization point data types are changed. In particular it is not necessary to edit the same section of source code, so one avoids the attendant risks of one customization overwriting another one, and difficulties in upgrading that section of source code.

The chief *disadvantages* of this approach to customization are that it requires foresight as to which customization points may be needed, and that multiple serial customizations of the same class cannot be developed independently of each other: one customization must be a subclass of the other customization, and hence must be aware of the existence of that other customization.

5.2 Information Hiding Using Interfaces

Interfaces allow one to hide some of the design decisions that are not relevant to clients. Since implementation details are unknown to clients, they do not become dependent on them, and it is much easier to evolve the specific implementation – hence the popular slogan, "Program to an interface, not to an implementation" [13]. Also, by combining information hiding and inheritance, programmers can extend existing interfaces in a subtype with new operations without breaking existing clients. This is the traditional approach to evolution in a object-oriented setting.

Even if interfaces support evolution of their implementations, one has to keep in mind that the interfaces themselves may need to evolve. Even if some design decisions can be hidden behind an interface, as proposed by Parnas [32], the published interfaces themselves cannot be changed without taking the risk of breaking a large number of external software systems that depend on them. An apparently harmless modification, such as adding a new operation to an interface in C#, can cause great trouble: all the existing classes that implement the previous version of the interface will have to be modified to support the new operation. Abstract classes, as found for example in Java and C#, are more interesting in this respect as they can sometimes meaningfully provide a default implementation for a new operation. Consider the following abstract class `Invoice`:

```
public abstract class Invoice {
  public abstract ICollection<Item> Items { get; }
}
```

It is possible to add a method `GrandTotal` to this abstract class without breaking the existing concrete subclasses:

```
public abstract class Invoice {
  public abstract ICollection<Item> Items { get; }

  public virtual double GrandTotal() {
    return Items.Sum(item => item.Price * item.Quantity);
  }
}
```

Note that if there is already a (non-virtual) method with the same name in the subclass, the compiler will give a warning that the subclass implementation of `GrandTotal` hides the inherited member. Note also that the default implementation provided by `Invoice` can be sub-optimal. For example a subclass that maintains the current total in an instance variable will gain from overriding `GrandTotal` and directly returning the instance variable.

```
public class InvoiceImp : Invoice {
  ...
  public override double GrandTotal() {
    return currentTotal; // instance variable
  }
}
```

The problem with abstract classes is that a class can only have one base class (in Java and C#), whereas it can implement multiple interfaces. This is not the case for languages that support multiple inheritance. But multiple inheritance tends to be criticized for its complexity and the problems that it brings along – such as the infamous diamond inheritance problem.

The Component Object Model (COM) [36] uses interfaces to support evolution of components as well as client programs. A component can be used only through its functions (operations, methods) as originally advocated by Parnas [31]. An interface is a set of functions, where each function is described by its signature: its name, its parameters (number, order and types), and its return type.

The following restrictions on COM components and their interfaces help mitigate evolution problems:

- An interface (with a given interface identifier) must remain forever unchanged once it has been published.
- A component may support any number of interfaces, and the set of interfaces it supports may change over time.
- A client program can, at runtime, ask a component whether it supports a particular interface (using its interface identifier) and hence whether the component supports particular methods.

The restrictions support evolution of components, because an updated component may exhibit new functionality through an additional interface, while continuing to support its old interfaces. The updated component will continue to work with existing client code, because such code will continue to ask the component for its old interface and will be unaffected by new functionality.

The restrictions also support evolution of the client code. Obviously, any change to the client that does not require new component behavior, will just work with old and new components alike. If a client is updated so that it would prefer to get some new behavior from a component, but can work with old client behavior (only less efficiently, say), then the updated client simply asks the component whether it supports the most desirable new interface that exhibits new behavior, and failing that, asks it whether it supports the second-most desirable interface, and so on. Hence this supports any number of steps of evolution.

If an updated component stops supporting some functionality (for instance, because it has been deprecated for security reasons), it will have to stop supporting some old interface. Client code will discover that at runtime when asking for the interface. Depending on the robustness of the client design, and the amount of foresight that went into the design of the interfaces, the client may be able to fall back on some other interface supported by the component; if not, it must give up.

The latter scenario shows one drawback of the COM model: mismatches in component evolution will not be discovered at compile time or deployment time, only at runtime, when the client asks the component whether it supports the requisite interfaces.

- *Need to Anticipate Customizations.* Following the concepts of information hiding, the designer has to come up with a list of design decisions which are likely to change. Hence there is a strong requirement to anticipate changes.
- *Control over Customizations.* One of the famous epigrams by Perlis [33] reads: "Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication". Types allow for a limited form of checking. Contracts, mentioned previously, are sometimes used to extend checking – but most of the control over customizations is typically done at design-time, through the use of static type checking.
- *Resilience to Kernel Evolution.* As long as the new version of the kernel conforms to the published interface, the program will still compile. Of course more guarantees than just type-conformance are typically needed to ensure correctness of the software system (as explained in the criteria section 4).
- *Support for Multiple Customizations.* There is no direct support for independently developed customizations, since the implementation of a specific interface is provided by a single class. Using a combination of inheritance and information hiding would allow for multiple sequential customizations (in the context of single inheritance), but using information hiding alone will not.

5.3 Parametric Polymorphism

Parametric polymorphism supports evolution because it can decouple some design decisions. For example, the designer of a new class `Stack<T>` will not have to foresee the possible kinds of elements that will be contained in the stack, and yet can enjoy type safety. Without parametric polymorphism, the designer of the class `Stack` would have to either make a new version of the class for each possible kind of element contained, such as `StackOfPerson`, `StackOfInt`, and so on, or he would have to compromise type safety by losing type information and using type casts, as in `Person p = (Person)myStack.Top`.

However, with parametric polymorphism or generic types as in Java, C# and ML, the behaviour of a parametrized type or method is the same for all type parameter instances — as implied by the term "parametric". Hence parametric polymorphism may support evolution but not really behavioural customization. This is in contrast to templates in C++ [37] and polytypic programming and generalized abstract data types in Haskell and extensions of C# [20], but we shall say no more about those mechanisms here.

- *Need to Anticipate Customizations.* In the previous `Stack` example, parametric polymorphism does not depend on anticipation of customization of the classes of the various element that will be stored in the stack – if the class `Person` changes, the class `Stack` does not have to change. But very often, we have to do more that just storing and retrieving objects from a collection: we need to use constraints on the formal generic types. For example if a class `Invoice` is seen as container of priced items, it is reasonable to require the first generic type to be constrained by an interface `IPriced`. But if such a

constraint is used on the formal type parameter, then we are back on the some problem as for information hiding: the interface `IPriced` can evolve. (Also one should note that the choice of using a generic type for a specific type declaration represents a form of anticipation itself.) For example, using C#:

```
public interface IPriced { double Price { get; }   }
public class Invoice<T> : Stack<T> where T : IPriced { ... }
```

- *Control over Customizations.* Similarly to other language based techniques presented above, the type system ensures some degree of correctness. The control over customizations is performed at design-time.
- *Resilience to Kernel Evolution.* A class `Stack<T>` with an unconstrained type parameter, as above, need not change when the item type `T` changes. However, a generic type `Painting<U> where U : Drawable` with a constrained type parameter `U` may need to change to be applicable to a new argument type.
- *Support for Multiple Customizations.* Parametric classes can have several formal type parameters, each of which can act as a placeholder until a runtime type is used [1, page 76]. One could devise a solution where each of these placeholders is used for a different customization.

5.4 Synchronous Events

In C#, so-called synchronous events, or callbacks, provide a flexible way to customize behaviour when one can foresee where customizations are needed. To add a customization point, one first declares a suitable delegate type (that is, function type), such as `After`:

```
public delegate void After(ref double result);
```

Then to prepare a class for customizations, we add an event field such as `after` to the class, and insert a conditional call to that event at the customization point:

```
public class Invoice {
  public static event After after;

  public double GrandTotal(int input) {
    double total = ...;
    if (after != null)
      after(ref total);              // Event raised here
    return total;
  }
}
```

Now assume we need a customization to give a 5 per cent discount on invoices over 10,000 Euros. The customization is added as a suitable anonymous method to the static event field of the `Invoice` class:

```
Invoice invoice = new Invoice();
Invoice.after += delegate(ref double result) {
  if (result >= 10000)
    result *= 0.95;
};
```

When the `GrandTotal` method of the `Invoice` class reaches the customization point, it will raise the event and call the anonymous method, which will reduce the `total` variable by 5 percent if it exceeds 10,000 Euros.

In the above example we associated the event with the class (as a `static` field) and hence obtain class-level customizability as in the object-oriented approach in section 5.1. Alternatively, one might use an instance field to obtain instance-level customizability.

- *Need to Anticipate Customizations.* There is a strong need to anticipate customizations, because one must create the necessary events and raise each event at all appropriate places, in the right order. Also, the type of the event being sent requires some insight into the forthcoming customizations.
- *Control over Customizations.* On one hand, the event argument types impose restrictions that support design-time control over customizations. One the other hand, triggering of events can be turned off at run-time providing a form of run-time control over customizations.
- *Resilience to Kernel Evolution.* The event model is quite fragile under changes to the base program: existing events may have to be raised at more or fewer places.
- *Support for Multiple Customizations.* Multiple customizations can be made simply by attaching multiple event handlers, so simultaneous development of customizations is straightforward. This of course does not prevent unwanted interactions between customizations as mentioned in the criteria section 4. Moreover, the order of event handler invocation may be significant, yet it may not be feasible to control the order in which handlers are invoked.

The chief *disadvantage* is that the event model is very dynamic—events can be attached and removed at runtime—so it is difficult to determine statically the properties of a system built with event listeners.

A less obvious disadvantage is that it is difficult to provide a complete specification of the contract between the listened-to object (the one raising the event) and the listening objects (those installing the event handlers). Namely, the installation `y.Event+=x.h` of an event handler `x.h` on object `y` is the beginning of a potentially long-lasting interaction between objects `x` and `y`.

Hence to understand and correctly use an event model, one must consider at least the following questions:

- What data can an event handler read, and what data can it modify? In Microsoft's Windows Forms framework, unlike Java's Abstract Window Toolkit, it is customary to pass the entire "sender" object `y` to the event handler, which seems to invite abuse by the event handler.

- What can the event handler assume about the consistency of data in the sender `y` when it is called, and what must it guarantee about the state of data in `y` when it returns?
- Could an event handler, directly or indirectly, call operations that would cause further events to be raised, and potentially lead to an infinite chain of events?
- At what points should an event be raised? This central design decision should be based on semantic considerations, since it strongly influences the correctness of upgrades of the kernel. For instance, it is better to specify that "the event is raised after a change to the account's balance" than to say that "the event is raised after one of the methods `Deposit` or `Withdraw` has been called". The former gives better guidance when new methods are added, or when considering bulk transactions such as `DepositAll(double[])` whose argument may be an empty array and hence perform no change to the account at all.
- What is guaranteed about multiplicity and uniqueness of events? For instance, consider a class Customer derived from class Entity, where method `Customer.M()` calls `base.M()`, and both implement an interface method specified to raise some event `E`. Should a call to `Customer.M()` raise the event once or twice?

5.5 Partial Methods as Statically Bound Events

The partial types and partial methods of the C# 3.0 programming language offer a statically bound alternative to events. Wherever there would be a call to an event handler, a call to a partial method is made instead. For instance, we may declare a partial method called `after` and call it as in this example:

```
public partial class Invoice {
  partial void after(ref double result);

  public double GrandTotal(int input) {
    double total = input * 1.42;
    after(ref total);
    return total;
  }
}
```

If the method call is needed, that is, if there is a customization at the call point, the partial method's body may be declared in a different source file:

```
public partial class Invoice {
  partial void after(ref double result) {
    if (result >= 10000)
      result *= 0.95;
  }
}
```

Then the two source files simply have to be compiled together, like this:

```
csc PartialMethod.cs PartialAfter.cs
```

If no customization is needed at the `after(...)` call point, one simply leaves out the `PartialAfter.cs` file when compiling `PartialMethod.cs`, and then the `after(...)` call will be ignored completely.

- *Need to Anticipate Customizations.* There is a strong need to anticipate customization points, because one must create the necessary partial methods and call them at all appropriate places.
- *Control over Customizations.* The partial method argument types impose restrictions that supports control of customizations at design-time to some degree.
- *Resilience to Kernel Evolution.* Similarly to events, the partial method customization model is rather fragile under changes to the base program: existing partial methods may have to be raised at more or fewer places.
- *Support for Multiple Customizations.* Partial methods offer no explicit support for multiple customizations since there can be only one implementation of a given partial method.

The chief disadvantage of partial methods, however, is that they are not dynamically configurable; unlike events they cannot be added and removed at runtime under program control. This provides poor support for the fluid way in which developers prefer to interact with e.g. Dynamics NAV, mentioned in section 3.6.

There is a position between that of dynamically-bound events that may be added and removed under program control (section 5.4) on the one hand, and the partial methods that require recompiling and reloading the application (as described above) on the other hand. Namely, one may use metadata to specify the association of event handlers with events, and prevent the running program from changing this association. This is the approach taken by Dynamics NAV. The approach would enable the development environment to tell which event handlers may be executed when raising a given event, and to discover potential event cycles by analyzing the metadata and the code of the event handlers. However, the other concerns and questions about events listed in section 5.4 must still be addressed.

5.6 Mixins and Traits

A mixin provides certain functionalities to the classes that inherit from it. It is sometimes said that the mixin "export its services" to the child class. When mixin composition is implemented using inheritance, mixins are composed linearly. Ducasse et al. [11] report several problems traditionally associated with mixins. For example, it is reported that class hierarchies are often fragile to changes since simple changes may impact many parts of the hierarchy. Traits can be seen as an attempt to solve some of the problems caused by mixins. A trait is, simply, a set of methods. A trait is not coupled with the class

hierarchy. Traits can be composed in arbitrary order (in their original definition) and can be used to increment the behavior of an existing class. Ducasse et al. emphasize that, using traits, the two roles of "unit of reuse" and "generator of instances" can be respectively assumed by traits and classes, whereas both roles are traditionally assumed by classes in object-oriented languages [11]. And since traits are divorced from the class hierarchy, they do not suffer from the problems associated with multiple inheritance.

Scala uses both mixins and traits to solve the code reuse limitations posed by single inheritance [29]. Its mixin class composition mechanism allows for the reuse of the delta of a class definition. The following example defines a trait `Invoice` with an abstract method `GrandTotal`. The class `InvoiceImpl` will provide the implementation for this abstract method. Note that the two are, for now, completely unrelated: `Invoice` and `InvoiceImpl` can be compiled independently. For the sake of simplicity for the example, the method implementation returns a constant.

```
trait Invoice {
  def GrandTotal: double              // Abstract definition
}
class InvoiceImpl {
  def GrandTotal: double = 10         // Candidate implementation
}
```

A different developer (for example, in a partner company), can provide a customization of the method `GrandTotal`.

```
trait DiscountInvoice extends Invoice  {
  abstract override def GrandTotal: double = super.GrandTotal * 0.95
}
```

Note that the developer implementing this customization does not have to know about the concrete implementation; his customization extends the trait `Invoice` and not the implementation class `InvoiceImpl`. Method `GrandTotal` is declared above as `abstract` since it overrides a method which is not defined. Similarly, another developer, (e.g., at another partner company), can define another customization implementing a simple 1 Euro tax rule:

```
trait OneEuroTax extends Invoice {
  abstract override def GrandTotal: double = super.GrandTotal + 1
}
```

Finally, a customer might want to combine the implementation `InvoiceImpl` with the two traits `DiscountInvoice` and `OneEuroTax` that customize the behavior of `GrandTotal`:

```
class DiscountFirst extends InvoiceImpl
with DiscountInvoice
with OneEuroTax
```

```
object Test {
  def main(args : Array[String]) : Unit = {
     // (10 * 0.95) + 1
     println("Total " + (new DiscountFirst).GrandTotal)
  }
}
```

Note that in this particular example, the order of the `with` clauses is significant, due to the linearization of the super calls. In this case, the discount will first be applied on the grand total, and then the one Euro tax will be added.

One of the problem with traits is that they usually do not give direct support for state. Traits must be stateless, which imposes some strict limitations on their use. Note that the traits community is actively working on stateful traits but the current proposals also have some limitations (instance variables are local to the scope of traits, with some exceptions), see [9].

- *Need to Anticipate Customizations.* Traits are attractive in our case since they allow for fine-granularity code reuse. But some foresight is required to design the collection of traits in a way that will be be most convenient for the person performing the customizations, especially the specific grouping of methods into traits.
- *Control over Customizations.* The compiler ensures type correctness. Using traits, the control over customizations is performed at design-time.
- *Resilience to Kernel Evolution.* We showed in our example that the customizations are decoupled from `InvoiceImpl` since they do not even need to know about its existence. One the other hand, if the base trait `Invoice` changes, the customizations will have to be adapted.
- *Support for Multiple Customizations.* The previous example demonstrated that `InvoiceImpl`, `DiscountInvoice` and `OneEuroTax` can all be developed independently, and finally composed together by the end-developer.

5.7 Aspect-Oriented Programming

Aspect-oriented programming [21] provides an alternative to the event models described in sections 5.4 and 5.5. Although some realizations of aspect-oriented programming restrict the insertion of extra code to the beginning or end of a method body, others allow code to be inserted at arbitrary (but previously identified) places in a method body [12]. Clearly the latter is equivalent to raising events at those places in the method.

One concern that speaks against this approach is that a well-designed method should encapsulate a state change that results in a coherent object state, so it seems to go against software engineering principles to permit arbitrary modifications to a method's body. This concern is similar to the concern that an event handler should not modify the event sender object in arbitrary ways; see section 5.4.

Here we consider only a rather special case of aspect-oriented programming, namely aspect-like static program rewriting. We use Yiihaw, a static aspect

weaver for C# that works by rewriting of bytecode files [18]. It reduces runtime overhead relative to event-based customization and permits static checks. However, while Yiihaw's pointcut language permits some quantification, it is not particularly expressive. Other aspect weavers, such as AspectJ [22], would provide more fine-grained customization, which would be an advantage compared to event-based customization.

Customization Using Aspects. Consider again customization of the Invoice example already seen in sections 5.1 and 5.4. Assume the `Invoice` class is declared on a lower layer with an instance method `GrandTotal`:

```
public class Invoice {
  public virtual double GrandTotal() {
    double total = ...;
    return total;
  }
  ... other members ...
}
```

As before, assume that at the higher layer we want to customize this to give a discount when the grand total exceeds 10,000 Euros. To do this, we separately declare an advice method as follows:

```
public class MyInvoiceAspect {
  public double DoDiscountAspect() {
    double total = JoinPointContext.Proceed<double>();
                                        // Customization point
    return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

and compile it, and then write an interception pointcut:

```
around * * double Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

The target assembly and the advice assembly are compiled using the C# compiler and then woven by an aspect weaver. In the resulting woven assembly, the `GrandTotal` method of the `Invoice` class will behave as if declared like this:

```
public class Invoice {
  public virtual double GrandTotal() {
    ... complicated code ...
    return total * (total < 10000 ? 1.0 : 0.95);
  }
  ... other members ...
}
```

The resulting woven method has the exact same signature as the original target method.

Sequential Customization by Further Weaving. The woven method can be used as target for further weaving. For instance, we may want to further modify the `Invoice` class and its `GrandTotal` method to count the number of times the `GrandTotal` method has been called. This involves adding a field `int count` to the class and making further advice on the method.

The additional pointcut file must contain an introduction and an interception:

```
insert field private instance int MyNewInvoiceAspect:count
  into Invoice;
around * * double Invoice:GrandTotal()
  do MyNewInvoiceAspect:DoCountAspect;
```

We need to declare an advice class with a field and an advice method as follows:

```
public class MyNewInvoiceAspect {
  private int count;
  public double DoCountAspect() {
    count++;
    return JoinPointContext.Proceed<double>();
  }
}
```

After compiling the advice and weaving it into the previously woven assembly, we get a class `Invoice` that will behave as if declared like this:

```
public class Invoice {
  private int count;
  public virtual double GrandTotal() {
    count++;
    ... complicated code ...
    return total * (total < 10000 ? 1.0 : 0.95);
  }
  ... other members ...
}
```

Evaluation of Aspects for Customization

- *Need to Anticipate Customizations.* Aspect-orientation does not require foresight as to where events need to be raised, but there is an analogous though less stringent need for foresight. Namely, customization points must be expressible as join points. In the case where only "around" interceptions are expressible, foresight is needed to factorize the kernel so that all customization points are methods, but it is not necessary to foresee *which ones* will be customized.
- *Control over Customizations.* The type system of the implementation language, combined with weave-time checks performed by the aspect weaver, give some assurance that customizations are meaningful, and can point out incompatible changes when one attempts to upgrade the base system.

- *Resilience to Kernel Evolution.* Aspect-oriented customization is fairly insensitive to evolution of the base code so long as the names and parameters of methods remain unchanged. However, if customized methods or their parameters get renamed, then the weaving may fail to customize a method it should have, or may wrongly customize one that it should not.
- *Support for Multiple Customizations.* Aspect-oriented customization supports independently developed customizations just as well as do events.

Some research indicates that an aspect approach to cross-cutting concerns makes software evolution harder, not easier, at least based on theoretical considerations [39]. It is not clear that those results extend to our use of aspects. When using aspects for cross-cutting concerns, join points are likely to be described by quantification, using only few pointcuts. However, when customizing software products, the join points are customization points and are more likely to be explicitly enumerated, using many pointcuts. Which gives more resilience to evolution is unclear.

Aspects for Customization in Dynamics AX. Static aspect weaving, as outlined above, offers a plausible way to perform customization of Dynamics AX applications (section 3):

- It preserves the layer model of Dynamics AX. This in turn offers several advantages. First, the overall philosophy will be readily understandable to the current developers at the Dynamics core development team, as well at partners and customers. Second, there is a likely upgrade path from the current AX implementation to an AX implementation based on layers and aspects.
- The aspect weaver can check, at weave time, the consistency of the modifications of upper layers with lower layers.
- Aspects can be statically woven so that they incur no performance penalty at all, and hence would perform no worse than the existing source code based customizations.

To express customizations as aspects we have used the Yiihaw aspect weaver [19] described by another paper in this volume [18]. Although several aspect weavers for .NET have been proposed, Yiihaw seems to be especially suited: it introduces no runtime overhead at all, it statically checks aspect code ahead of weave-time, it statically checks consistency of weaving, and it can further weave an already woven assembly as indicated above. This is necessary in the Dynamics AX scenario where lower layer code gets customized in a higher layer, and the result gets further customized in an even higher layer; see figure 1 on page 225. The limitations of the Yiihaw pointcut language and its notion of aspect mean that some will consider it a tool for feature composition rather than a full-blown aspect weaver, but it seems adequate for the purposes considered here.

5.8 Software Product Lines Using AHEAD

Feature-oriented programming has been developed over many years by Batory and coworkers [7,8,6,35]. Part of the motivation for this work is the insight that

future software development techniques will synthesize code and related artifacts (such as documentation) extensively. The research efforts have focused on structural manipulation of these artifacts. These ideas can be seen as part of the metaprogramming research field: programs are treated as data, and transformations are used to map programs to programs.

These ideas gave rise to concrete tools, among which GenVoca and AHEAD [5] are prime examples. These tools were used to synthesize product lines for various domains such as database systems and graph libraries. More concretely, using a product line, a user can select among a set of predefined features and the tool will combine artifacts to generate a program that implements the desired functionality. The user typically uses a declarative domain-specific language to express the feature selection he wants.

Among the various artifacts handled by these tools we henceforth focus our attention on source code. The mixin is one of the core object-oriented concepts that underpin this approach to code composition. In this context, a mixin is a class whose superclass is specified as a parameter. Using the variant of Java proposed by AHEAD, we can write a customization for the invoice example from section 5.6:

```
layer tax;
refines class invoice {
  overrides public double grandTotal() {
    return Super().grandTotal() + 1;
  }
}
```

This customization adds one Euro, a "tax", to the grand total computed in the base code (omitted for the sake of brevity). Note that the customization is defined in a named layer "tax". The discount customization, that we saw previously, can be programmed similarly in a layer "discount". The discount is unconditional in this case to make the example a bit shorter.

```
layer discount;
refines class invoice   {
  overrides public double grandTotal() {
    return Super().grandTotal() * 0.95;
  }
}
```

To compose the base code `invoice` with the customizations, the programmer can choose between two tools. The first one, called "mixin", will transform the composition into a class hierarchy. Using this tool, each customization will be turned into an abstract class that extends another abstract class, with the exception of the last customization, `discount` in our case, which is turned into a concrete class. Each class name in the hierarchy is a mangling of the name `invoice` with the name of the originating layer – again with the exception of the class that corresponds to the last customization (since it is the one that will be instantiated).

```
package invoice;
abstract class invoice$$invoice implements invoice  {
  public double grandTotal() {
    return ...;
  }
}
abstract class invoice$$tax extends  invoice$$invoice    {
  public double grandTotal() {
    return super.grandTotal() + 1;
  }
}
public class invoice extends  invoice$$tax    {
  public double grandTotal() {
    return super.grandTotal() * 0.95;
  }
}
```

The other tool, called "jampack", offers a more compact encoding of the code composition. In this case, the base code and the customizations are turned into static methods, with the exception of the last customization which is mapped into a non-static method. The name mangling for method names is very similar to the name mangling for class names performed by the other tool.

```
package invoice;
public class invoice {
  public final double grandTotal$$invoice() {
    return ...;
  }
  public final double grandTotal$$tax() {
    return grandTotal$$invoice() + 1;
  }
  public double grandTotal() {
    return grandTotal$$tax() * 0.95;
  }
}
```

Mixins are often not conceived in isolation, but rather "carefully designed with other mixins and base classes so that they are compatible" [5]. It is easy to see in the above example that overriding `grandTotal` might break some other code that relies on its initial semantics.

A particularly interesting feature of this work is the composition algebra and design rule checking. The design rules are necessarily domain-specific, for instance, for the domain of efficient data structures. Batory's feature-oriented programming for product lines [4] seems highly relevant and makes many points of value for evolvable software products.

– *Need to Anticipate Customizations.* Similarly to classical object-oriented programming, it seems that product-line engineering requires that the programmer has a good understanding of the domain. Classes must be designed in such way to accommodate for mixin composition conveniently.

- *Control over Customizations.* The AHEAD tools suite will check that the types are conforming, but no guarantee is given on the semantics. It is up to the designer to ensure that the prescribed composition of code artifacts is meaningful for the domain.
- *Resilience to Kernel Evolution.* If we assume the closed-world assumption that is common within software product lines, all the potential customizations and their possible interactions are known. Therefore an evolved kernel can be organized in such way that any existing choice of features will continue to work as intended. This does not mean that upgradability comes for free: the kernel developer must understand these interactions and handle them.
- *Support for Multiple Customizations.* The product line is the family of classes created by mixin composition. As noted before, the mixin approach requires that mixins are not created in isolation, but rather carefully designed together, which basically assumes a closed world of possible customizations. Therefore there is no support for independently developed customizations.

5.9 Software Product Lines Using Multi-dimensional Separation of Concerns

The Hyper/J framework and tool developed by Tarr, Ossher and others at IBM Research [30] support multi-dimensional separation and integration of concerns in Java programs, which may be used to implement software product lines. A Hyper/J prototype implementation [16] is publicly available, but is not currently actively supported. In particular, the prototype does not seem to work with the latest version of the Java runtime environment, which seriously limits its usability. Hyper/J shares many goals with aspect-oriented programming, such as the decomposition of software systems into modules, each of which deals with a particular concern.

A *dimension of concern* is a class, a feature, or a software artifact. For example, a class in a code base represents a class concern. Each dimension of concern gives a different approach to software decomposition. Tarr and others coined the term "the tyranny of the dominant decomposition" to signify that a programming language typically supports only one (dominant) decomposition, such as classes in case of object-oriented languages. Consequently some concerns cannot be implemented in a modular manner, and the code fragments implementing them will be scattered across the modules that arose from the dominant decomposition [30, page 5]. For instance, logging (of method calls) is an example of such as cross-cutting concern, often cited in aspect-oriented programming.

Using Hyper/J, decomposition can be done simultaneously along multiple dimensions of concern: The class is no longer the main decomposition mechanism in an object-oriented language, putting class, package, and functional decomposition on a more equal footing. The Hyper/J tool takes care of the interaction across those different decompositions. The goal is to encapsulate into new modules those concerns that were previously scattered over the classes.

By combining selected concerns into a program, a programmer can create a version of the software containing only selected features, even if the original software system was not written with separation of features in mind [30].

Units are organized in a multi-dimensional matrix, where each axis is a dimension of concern, and each point on the axis is a concern in that dimension. The main units in Hyper/J are functions, class variables, and packages. Concern specifications are used to specify the coordinate of each unit within the matrix, using the notation:

```
x: y.z
```

where `x` is a unit name, `y` a dimension and `z` a concern.

We now give a Hyper/J solution to the invoice example from section 5.6. Once again there is a base implementation of `Invoice`, now in Java. The method `GrandTotal` computes the sum of the items of the invoice, and another method called `GetTotal` will return that total.

```
package lipari.base;
public class Invoice {
  private double total;
  public void GetTotal() {
    return total;
  }

  public void GrandTotal() {
     total = 10; // Dummy implementation
  }
}
```

In another package, a developer defines a discount as a customization of the base implementation by writing the following class:

```
package lipari.discount;
public class Invoice {
  double total;

  public double GrandTotal(double x) {
    total = total * 0.95;
  }
}
```

Note that the name used for the method and for the instance variable mimic the ones from the base code, but the package name is different. The "one Euro tax customization" can be specified similarly to the discount customization above, in a separate package. Note that both the customizations and the base class can be compiled completely independently.

A programmer can then compose the base code with the two customizations by writing the following Hyper/J specification (some parts were omitted for brevity). First, we specify the concerns:

```
-concerns
  package lipari.base : Feature.Base
  package lipari.tax :  Feature.Tax
  package lipari.discount : Feature.Discount
```

In this case the mapping is simple since each concern is implemented by its own package. Then we specify that we want to compose a software system, here called `LipariHypermodule`, using the concerns specified above:

```
-hypermodules
  hypermodule LipariHypermodule
  hyperslices: Feature.Base, Feature.Discount, Feature.Tax;
  relationships:
    mergeByName;
    merge class  Feature.Base.Invoice,
                 Feature.Discount.Invoice,
                 Feature.Tax.Invoice;
    end hypermodule;
```

Note the composition relationship `mergeByName`, which indicates that units in different hyperslices that have the same name will be fused. Using the composition specification above, the tool can generate a new software system with the selected features. The code below will correctly display the expected total, 10 Euros with a 5% discount, followed by a one Euro tax – that is, 10.5 Euros.

```
package lipari.base;
public class Main {
    public static void main(String[] args) {
        lipari.base.Invoice i = new lipari.base.Invoice();
        i.GrandTotal();
        System.out.println("Total = " + i.GetTotal() );
    }
}
```

- *Need to Anticipate Customizations.* Some foresight is required to identify the dimensions of concern because they determine how concerns can be combined into systems. It seems that concerns may be added to a dimension as needed.
- *Control over Customizations.* Types provide some protection against meaningless compositions at design-time.
- *Resilience to Kernel Evolution.* If we have a closed-world assumption, similarly to what was mentioned in section 5.8, the evolution of the kernel can be done in such way that any existing choice of features continue to work. Of course, the same constraints mentioned in section 5.8 apply here.
- *Support for Multiple Customizations.* Again as it was mentioned before, under a close-world assumption there is no support for other independently developed customizations other than those that could be foreseen when designing the kernel.

Table 1. Summary evaluation of customization technologies. Legend: Need to Anticipate Customizations: (1) none, (2) customization points, (3) customization kinds. Control over Customizations: (a) design-time control, (b) run-time control, (c) none. Resilience to Kernel Evolution: (i) some resilience, (ii) restricted resilience, (iii) no resilience. Support for Multiple Customizations: (I) for parallel development, (II) for sequential development, (III) no support.

Technique	Sec.	Impl.	Refs.	Need to Anticipate Customizations	Control over Customizations	Resilience to Kernel Evolution	Support for Multiple Customizations
Inheritance	5.1	C#	[1]	(2)	(a)	(ii)	(II)
Inform. hiding	5.2	C#	[32,1]	(2)	(a)	(ii)	(III)
Param. polymorphism	5.3	C#	[1]	(2)	(a)	(ii)	(II)
Events	5.4	C#	[1]	(2)	(a) and (b)	(ii)	(I)
Partial methods	5.5	C#	[1]	(2)	(a)	(ii)	(III)
Mixins, traits	5.6	Scala	[11,29]	(2)	(a)	(ii)	(I)
Aspects	5.7	Yiihaw	[19]	(1)	(c)	(iii)	(I)
SPL using AHEAD	5.8	AHEAD	[5]	(3)	(a)	(i)	(III)
SPL using MSC	5.9	Hyper/J	[30]	(3)	(a)	(i)	(III)
AX layers	5.10	Dynamics	[14]	(1)	(a)	(iii)	(II)

5.10 The Dynamics AX Layer Model

The source-code based layered customization models of Dynamics AX was described in section 3.8. Here we just give a brief assessment of it for comparison with the other technologies surveyed in the following sections.

- *Need to Anticipate Customizations.* There is no need to anticipate customizations, since any lower layer application element can be copied to a higher layer and customized there.
- *Control over Customizations.* A customization can include any edits, so there is no support for controlling customizations.
- *Resilience to Kernel Evolution.* The customizations are very fragile to base program evolution; it is entirely up to the developer to identify what changes need to be made to the customizations.
- *Support for Multiple Customizations.* The support is very good if the changes are made sequentially, for instance, if a customized component is further customized at a higher layer.

5.11 Summary Evaluation

Table 1 summarizes the properties of the technologies surveyed.

6 Conclusion

We defined the *upgrade problem* as the conflict between customization and evolution of flexible software products. We have presented the Dynamics enterprise resource planning systems as prime examples of such software products, and discussed how they are structured and customized, underscoring that the upgrade problem is a real one and the focus of much attention also in industrial contexts.

We then considered a number of software technologies and practices that are traditionally used for customization and for creation of families of related software systems. For each one, we have given a description, an example, and an evaluation in relation to four criteria: need for foresight, control over customizations, resilience to kernel evolution, and support for multiple independent customizations.

A tentative conclusion of this investigation is that static aspects (in the Yiihaw guise [18]) and traits offer good static correctness guarantees and good support for independent customization. They fit well with the structure of Dynamics AX (section 3.9) but rely too much on build-time software composition to fit well with the development practices around the Dynamics NAV (section 3.6). Also, they both require some foresight in defining the customization points, which must be classes and methods, and they are rather fragile in case class names or method names in the kernel are changed as a consequence of kernel evolution.

Software product lines offer some interesting potential to deal with the upgrade problem but their closed-world assumption does not fit the domain of enterprise resource planning (ERP) systems that we took for a case study here. Such systems must be customizable to unforeseeable legislation and new business models, and this poses additional upgrade challenges.

Acknowledgements. Thanks to the anonymous referees whose comments led to many improvements and clarifications. This work is part of the project Designing Evolvable Software Products, sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen. For more information, see `http://www.itu.dk/research/sdg`.

References

1. C# language specification. ECMA Standard 334 (June 2005)
2. Allen, E.: Object-oriented programming in Fortress. FOOL/WOOD 2007, (January 2007), `http://www.cs.hmc.edu/`
3. Allen, E., et al.: The Fortress language specification. Technical report, Sun Microsystems (March 2008), `http://research.sun.com/projects/plrg/`

4. Batory, D.: Feature oriented programming for product-lines. Slide set for tutorial, OOPSL 2004, Vancouver, Canada (October 2004)
5. Batory, D.: Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Systems Journal 45(3), 527–539 (2006)
6. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS, tools for implementing domain specific languages. In: Fifth International Conference on Software Reuse, pp. 143–153 (1998)
7. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology 1(4), 355–398 (1992)
8. Batory, D., Singhal, V., Sirkin, M., Thomas, J.: Scalable software libraries. In: SIGSOFT, pp. 191–199 (1993)
9. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Computer Languages, Systems & Structures 34(2-3), 83–108 (2008)
10. Dittrich, Y., Vaucouleur, S.: Customization and upgrading of ERP systems. an empirical perspective. Technical Report TR-2008-105, IT University of Copenhagen, Denmark (March 2008)
11. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems 28(2), 331–388 (2006)
12. Eaddy, M., Aho, A.: Statement annotations for fine-grained advising. In: ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2006), Nantes, France, (July 2006)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
14. Greef, A., et al.: Inside Microsoft Dynamics AX 4.0. Microsoft Press (2006)
15. JSR-277 Expert Group. Jsr-277: Java module system. Technical report, Sun Microsystems (October 2006), `http://jcp.org/en/jsr/detail?id=277`
16. Hyper, J.: Home page, `http://www.alphaworks.ibm.com/tech/hyperj`
17. Software Engineering Institute.Software product lines, `http://www.sei.cmu.edu/productlines/`
18. Johansen, R., Sestoft, P., Spangenberg, S.: Zero-overhead composable aspects for .NET. In: Börger, E., Cisternino, A. (eds.) Software Engineering. LNCS, vol. 5316, pp. 185–215. Springer, Heidelberg (2008)
19. Johansen, R., Spangenberg, S.: Yiihaw. an aspect weaver for .NET. Master's thesis, IT University of Copenhagen, Denmark (February 2007), `http://www.itu.dk/people/sestoft/itu/JohansenSpangenberg-Aspects-2007.pdf`
20. Kennedy, A., Russo, C.: Generalized algebraic data types and object-oriented programming. In: OOPSLA, San Diego, California, October 2005, pp. 21–40 (2005)
21. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Lehman, M.M.: Rules and tools for software evolution planning and management. Annals of Software Engineering 11(1), 15–44 (2001)
24. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060–1076 (September 1980)

25. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a taxonomy of software evolution. In: International Workshop on Unanticipated Software Evolution, Warsaw, Poland (April 2003)
26. Microsoft. Microsoft Dynamics AX. Homepage, http://www.microsoft.com/dynamics/ax/
27. Microsoft. Microsoft Dynamics NAV. Homepage, http://www.microsoft.com/dynamics/nav/
28. Mortensen, F.: Software development with Navision. Talk, ERP Crash Course, University of Copenhagen, January 31 (2007), http://www.3gerp.org/Documents/ERP
29. Odersky, M.: The Scala language specification, version 2.0. Technical report, École Polytechnique Féderale de Lausanne, Switzerland (January 2007), http://www.scala-lang.org/
30. Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java. In: ICSE 2001: 23rd International Conference on Software Engineering, Toronto, Canada, pp. 821–822. IEEE Computer Society, Los Alamitos (2001)
31. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
32. Parnas, D.L.: On the design and development of program families. IEEE Transactions on Software Engineering SE2(1), (1976)
33. Perlis, A.J.: Epigrams on programming. SIGPLAN Notices 17(9), 7–13 (1982)
34. Pontoppidan, M.F.: Smart customizations. Screen cast (2006), http://channel9.msdn.com/Showforum.aspx?forumid=38&tagid=94
35. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
36. Rogerson, D.: Inside COM. Microsoft's Component Object Model. Microsoft Press (1997)
37. Stroustrup, B.: The C++ programming language. Addison-Wesley, Reading (2000)
38. D. Studebaker. Programming Microsoft Dynamics NAV. Packt Publishing (2007)
39. Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In: AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA (2003)