

# Influence of type systems on dynamic software evolution

*Yves Vandewoude*

*Peter Ebraert*

*Yolande Berbers*

*Theo D'Hondt*

*Report CW 415, June 2005*



Katholieke Universiteit Leuven  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Influence of type systems on dynamic software evolution

*Yves Vandewoude*

*Peter Ebraert*

*Yolande Berbers*

*Theo D'Hondt*

*Report CW415, June 2005*

Department of Computer Science, K.U.Leuven

## **Abstract**

Currently, no programming language exists that was specifically designed with dynamic software evolution in mind. Therefore, additional effort is required from developers to compensate for this lack of support. In this paper, we analyze and categorize the most important features a programming language should offer to adequately support dynamic evolution. We argue that many of these features are tightly coupled with the typesystem of the programming language and subsequently investigate the impact of the type system on the different aspects of dynamic software evolution. In addition, we also analyze the suitability of existing programming languages with respect to dynamic software evolution. The main contribution of our paper is a table that summarizes our findings and clearly shows that languages with weaker type system have more power to carry out runtime changes but at a cost of reduced safety. This table can be used by application developers to select a suitable language for their project or by language designers aiming to create a programming language for dynamic adaptation.

**Keywords :** Dynamic Evolution, Type Systems.

## Research

# Influence of type systems on dynamic software evolution



Yves Vandewoude<sup>1\*</sup>, Peter Ebraert<sup>2</sup>, Yolande Berbers<sup>1</sup>, Theo D'Hondt<sup>2</sup>

<sup>1</sup> *Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium*

<sup>2</sup> *Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium*

---

## SUMMARY

Currently, no programming language exists that was specifically designed with dynamic software evolution in mind. Therefore, additional effort is required from developers to compensate for this lack of support. In this paper, we analyze and categorize the most important features a programming language should offer to adequately support dynamic evolution. We argue that many of these features are tightly coupled with the typesystem of the programming language and subsequently investigate the impact of the type system on the different aspects of dynamic software evolution. In addition, we also analyze the suitability of existing programming languages with respect to dynamic software evolution. The main contribution of our paper is a table that summarizes our findings and clearly shows that languages with weaker type system have more power to carry out runtime changes but at a cost of reduced safety. This table can be used by application developers to select a suitable language for their project or by language designers aiming to create a programming language for dynamic adaptation.

KEY WORDS: Dynamic Evolution; Type Systems

## 1. Introduction

“If it’s not broken: don’t fix it.” This idiom is above all applicable to software. Nevertheless, software evolution is an enormous problem that makes up for more than 80% of the cost of a software system [Rausch, 2000]. During the lifecycle of a software system, many bugs

---

\*Correspondence to: Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Contract/grant sponsor: Yves Vandewoude and Peter Ebraert are supported by a scholarship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)

---



are discovered that must be fixed. In addition, even flawlessly working software often needs adaptation due to changing requirements. Despite many modern design technologies, evolving software is an extremely cumbersome process.

Well known techniques such as refactoring [Opdyke, 1992], [Fowler, 1999] attempt to lower the cost of evolution by continuously enhancing its structure (e.g. by reducing coupling) and making it more suitable for pending changes. Techniques for dynamic software evolution attempt to evolve a running system without shutting it down. This dramatically increases the complexity of the evolution problem since there are considerably more constraints on a running system. The updates must complete in a short timeframe, must deal with the state contained in the active application and consistency must be preserved during and after the change.

## 2. Problem statement

Currently more than two thousand programming languages already exist, each one different from the other. These programming languages can be grouped in categories with common characteristics. Some widely accepted taxonomies exist that define key properties of different programming languages (such as [Abelson and Sussman, 1985], [Scott, 2000]).

These taxonomies clearly show that there are no superior or inferior languages. Every language has its strengths and weaknesses and while most of them are general purpose in nature, they were all designed with a specific goal in mind: Scheme for educational purposes, SQL for database querying, C for system programming, Fortran for mathematical computations and Java for portability. These goals are clearly reflected in the design of the language as specific design decisions have been made to ensure that the primary goal could be optimally supported.

Being general purpose, most languages can be used outside the scope they were originally designed for. However, this often requires an additional effort from the programmer who needs to compensate for the lack of explicit support of a concept or feature. This is what currently happens for dynamic software evolution in which programs are changed at runtime. Languages that were not specifically designed to support runtime changes are used to implement adaptable software systems. Since no language support is provided for problems specific to dynamic change, programmers typically attempt to bypass them using a layered approach [Brodsky et al., 2001], [Ebraert and Tourwe, 2004], [Mens and Wermelinger, 2002], [Vandewoude and Berbers, 2004a], [Wuyts, 2001]. We claim that a programming language specifically designed with dynamic evolution in mind would resolve some of the more common problems and increase the practical applicability of adaptable software by providing features such as security and state consistency. Unfortunately, to the best of our knowledge, no such programming language has been developed so far.

The goal of our research is to establish the specification of programming languages that are well suited for dynamic software evolution. This specification can then be used (1) by software engineers to choose an appropriate language for developing software, and (2) by language engineers, to develop a new programming language towards dynamic software evolution.



Discussing every aspect of each programming language is not feasible. Therefore, based on the previously mentioned taxonomies of programming languages, this paper will use the type system of a language as its distinguishing feature. The reason for this choice is that the authors of this paper have all developed an evolution framework in languages with different type systems. Our experiences with these languages are therefore an excellent starting point for evaluation of their suitability in the field of software evolution. In addition, as will be shown, many other features of programming languages (such as reflection and safety) are directly related to their type system (or lack thereof).

In order to investigate the influence of the type system on different aspects of evolution, these aspects must be identified as well. For that purpose, a taxonomy previously published by Buckley et al. in the *Journal of Software Maintenance and Evolution: Research and Practice* [Buckley et al., 2005] is used.

The remainder of this paper is structured as follows. Section 3 discusses the different characteristics of typing in programming languages and their consequences. Section 4, presents a recently published taxonomy on software evolution and the different dimensions that it introduces [Buckley et al., 2005]. Section 5 represents the main contribution of this paper and will discuss the influence of a type system of a programming language on all the different dimensions of software evolution that were identified in the taxonomy by Buckley et al. The results of our findings are clearly summarized in table 5, in which the suitability of a language for dynamic evolution is related to its type system. In addition, we will use our findings to critically assess the taxonomy that was used, and to suggest possible improvements to this taxonomy. We illustrate the results from our analysis with a short real life example in section 6. We continue with related and future work (section 7) and conclude in section 8.

### 3. Type systems

This paper uses the type system of a programming language as its distinguishing feature. A type system itself, however, is not a trivial concept. Since it is of vital importance to establish accurate terminology for the remainder of this paper, this section introduces relevant concepts and definitions on type systems. According to [Scott, 2000], the four defining characteristics of a type system are *type binding*, *type checking*, *type conversions* and *type strength*.

#### 3.1. Type binding and type checking

**Definition 1 (Type Binding)** *Type binding is the process of assigning a type to a value. A distinction is made between static and dynamic binding depending on whether the binding occurs respectively at compile-time or at run-time.*

**Definition 2 (Type Checking)** *Type checking is the process that verifies whether the operands of an operator have compatible types. When an operator is applied to an operand of an incorrect type, a type error occurs. Depending on when the type checks occur, the term ‘static type checking’ or ‘dynamic type checking’ are used.*



		Type Checking		
		<i>Static</i>	<i>Both</i>	<i>Dynamic</i>
Type Binding	Static	Common: referred to as <i>statically typed</i> languages	Impossible	Impossible
	Both	Theoretically possible, but uncommon in real programming languages.	Common: referred to <i>statically &amp; dynamically typed</i> languages.	Theoretically possible, but uncommon in real programming languages.
	Dynamic	Impossible	Impossible	Common: referred to as <i>dynamically typed</i> languages.

Figure 1. Possible combinations of Type checking and Type binding

These definitions present two distinct but strongly related typing properties. For doing static checking, one needs static binding. This combination is referred to by the term *static typing*. This corresponds to the intuition which says that a language is statically typed if type information is present (and used) at compile time. With respect to these definitions, it is not relevant whether the type information is directly available through annotations (such as in Java or C#) or inferred (like in Haskell or ML). By analogy, dynamic checking requires dynamic binding. The term *dynamic typing* refers to a language that is both dynamically bound and dynamically checked. It is a common misconception that a programming language must be either statically or dynamically typed [Cartwright and Fagan, 1991]. It is vital to stress that static and dynamic typing are *not* mutually exclusive. Many programming languages combine both static and dynamic typing (e.g. Java, C#), other languages are neither statically nor dynamically typed (most assembly languages). The relationship between type binding and type checking is shown in figure 1. This figure also shows that although it is theoretically possible to have a programming language with both static and dynamic binding but only static or dynamic checking, this is rather uncommon in practice. Indeed, it makes little sense to include type information if that information is not used.



### 3.2. Type conversion

The term *type conversion* is used to describe the conversion of a variable from one type to another. Two different type conversions exist: implicit conversions (*coercions*) and explicit conversions (*castings*). The difference between both kinds of conversions lies in the annotation: whereas explicit conversions are always annotated, implicit conversions are detected and automatically added by the programming system.

### 3.3. Type strength

On the concept of type strength, there is much disagreement in literature. As Benjamin C. Pierce once stated: “*I spent a few weeks trying to sort out the terminology of strongly typed, statically typed, safe typed, etc., and found it amazingly difficult. The usage of these terms is so various as to render them almost useless.*” As such, very often, ad hoc definitions are given that are based on consequences of type strength. Many definitions in literature are contradictory, others are merely orthogonal [Finkel, 1996], [Schmidt, 1994], [Sethi, 1996], [MacLennan, 1986].

Since a terminology is required for the remainder of this paper we take the following definition, which was found to be a common denominator in most definitions:

**Definition 3 (Type Strength)** *Type strength is expressed by the terms strong and weak typing. They refer to the effectiveness with which a type system prevents type errors. A strongly typed language prevents any operation on the wrong type of data. In weakly typed languages there are ways to escape this restriction: type conversions.*

Figure 2 shows the relation between type strength and type safety. A type safe program cannot contain any type errors. The left part of figure 2 shows that weakly typed systems accept unsafe programs. Strongly typed systems only accept these programs for whom it can guarantee that there are no type errors. This is only a subset of all type safe programs (right portion of figure 2). Note that the notion of *accepting* a program does not exist if there is no type checking before execution, and thus is this notion non-existent in dynamically typed languages.

An extreme example is a type system that rejects every possible program. While evidently safe, it is also extremely useless. This is an example of the well known trade-off between type-safety on the one hand and language-power on the other hand [Sethi, 1996]. Note that a `ClassCastException` or `MessageNotUnderstood` are not considered to be type errors since their effect is defined by the language specification.

In addition, type strength is not entirely independent from type binding and type checking. This is clearly illustrated in the following example: the evolution of the Python programming language. Python is strongly and dynamically typed. Recently however, discussions within the community lean towards the inclusions of (optional) static annotations for Python as this would enhance the readability of the code and the compilation speed. Since these annotations are optional, type checking is only performed on these parts of the program that have been annotated. The new Python would therefore be weaker typed than the current language, since coercions would be necessary to combine annotated with unannotated code.

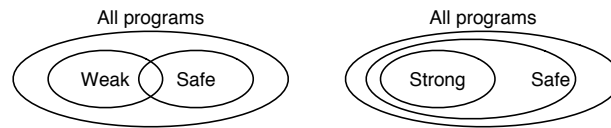


Figure 2. Weak versus Strong languages

### 3.4. Consequences of type systems: Reflective power vs. Safety

One of the main reasons that the type system of a language has a large impact on its dynamic evolution capabilities is because of the strong relationship between the type system and reflection. Reflection is the ability to inspect (introspection) and modify (intercession) the high-level structure of a program at runtime [Maes, 1987].

To realize full reflection, all system entities must be available as first-class entities which can be inspected or modified (in OO languages these entities are methods and objects). However, the reification of a runtime structure into a first-class entity requires that such a structure exists. This is only the case in the presence of dynamic typing.

Dynamic typing does not prevent the presence of a static type system. Nevertheless, static typing and reflection are two properties that do hinder each other. The first reason is that static typing is based on deduction of universal properties about a program's behavior before it is run. Reflection strongly complicates this, since it can be used to alter behavior based on run-time input, which is not available for prior analysis<sup>†</sup>[Alanko, 2004]. A second explanation of the clash between a static type system and reflection can be found in the fact that a static type system adds constraints that forbid certain changes to occur (all changes that violate type safety). Reflective systems for purely dynamically typed languages (such as Smalltalk) are therefore often more powerful than those for statically typed languages.

It is important to remark that for most statically typed languages, there is no fundamental technical issue in implementing reflective capabilities that violate their type safety. However, such an implementation would be in fundamental violation of the language itself (for an example: see section 5.2.2). In addition, we could conceive a statically typed language allowing dynamic violations of its static type system through reflection. However, to the best of our knowledge, such language does not exist, since designers of statically typed languages are not inclined to sacrifice the safety of their static typesystem. Safety is, after all, one of the main reasons to introduce static typing in the first place.

Figure 3 shows an overview of a few well known programming languages and their typing. The closer a language is located near the origin of figure 3, the higher its safety is. This increased safety comes at the cost of reduced flexibility and power. Concerning type strength,

<sup>†</sup>This is also why reflection is ignored in most type formalisms.

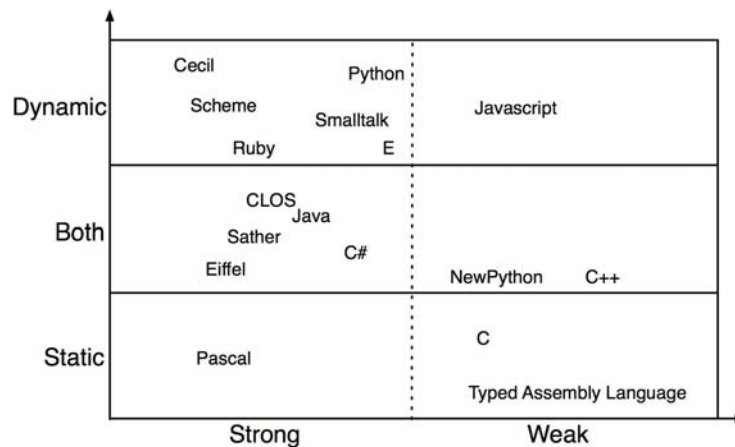


Figure 3. Prog. languages and their typing

the vertical dashed line stresses that there is no clear separation between strongly and weakly typed languages. As weak typing allows the programmer to escape restrictions of the type system using type conversions, programming languages with stronger type systems are located closer to the origin. A similar argument is made with respect to dynamic vs. static typing: purely dynamically typed languages are more flexible than languages with both static and dynamic typing, since their reflection mechanisms are not bound by language restrictions that are caused by their static typing. This explains their location at the very top of figure 3. When safety is an issue, safer languages are desired which are located closer to the origin. Full reflection on the other hand, requires type information at runtime. It can therefore only be provided by these languages that are either dynamically, or both statically and dynamically typed.

#### 4. Evolution Taxonomy

Directly analyzing the applicability of all different programming languages to runtime software evolution is not only unfeasible, it would also very quickly result in ideological discussions of specific language features. In this paper, we take the reverse approach and start from an existing taxonomy on evolution in which different aspects of dynamic software evolution are treated. As a base for discussion, we take the recently published taxonomy of software evolution presented in [Buckley et al., 2005].

It focuses on the mechanisms of a software change and the factors that influence these mechanisms, rather than the purpose of a change, which was the primary focus of earlier

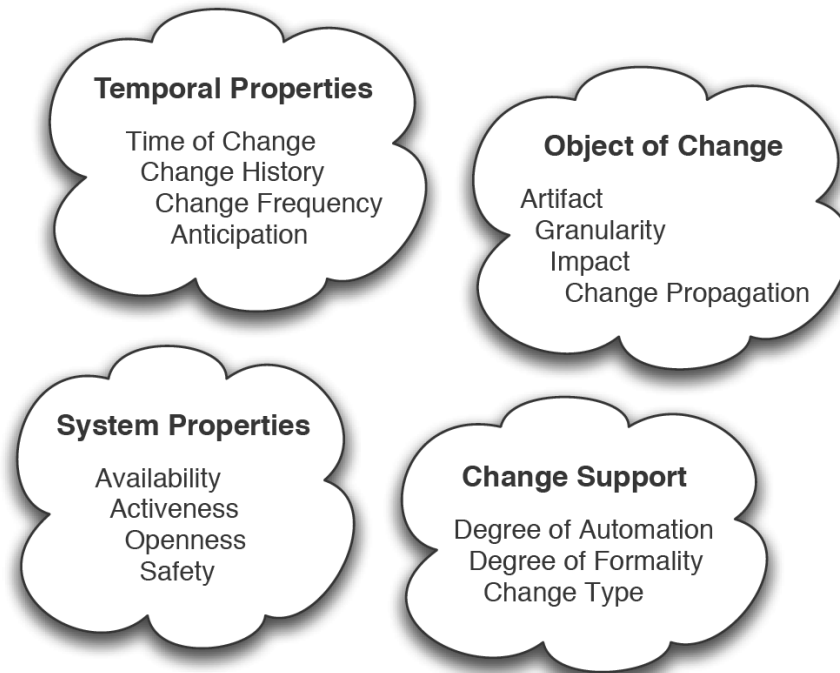


Figure 4. The Evolution taxonomy by Buckley et al. used in this paper (from [Buckley et al., 2005])

classifications. It is this clear focus on the technical characteristics of change mechanisms and its summary of different technological issues related to software evolution, that makes this taxonomy ideally suited for our purpose.

Its authors identify a number of dimensions as the building block of their taxonomy. Every characteristic that either actively influences the mechanism of a change, or which was often used in literature to characterize software evolution was taken as a dimension. The different dimensions are then grouped in 4 themes, which are logical groupings of the different dimensions: (i) temporal properties, (ii) the object of change, (iii) system properties and (iv) change support. The most important dimensions from [Buckley et al., 2005] and their grouping in the 4 themes are shown in figure 4.

The authors of [Buckley et al., 2005] clearly indicate that the themes are only one possible categorisation of the different dimensions, and that many other groupings are possible. However, although the grouping may influence the order in which the different dimensions are discussed, they are not vital for the main contribution of our paper. Since their grouping is a natural way of structuring the many dimensions, it will be followed in this paper as well.



---

## 5. Impact analysis of type systems on software evolution

Using the taxonomy introduced in the previous section as a red line, this section presents our analysis of the relation between a programming language's type system and software evolution. For each of the four themes shown in figure 4, we will discuss the different dimensions and their relation to the type system of a programming language. In addition, we will also touch very briefly on a fifth theme: the purpose of change. Although this theme was not explicitly treated in [Buckley et al., 2005], it has been discussed in depth in other publications such as [Lientz and Swanson, 1980] and [Chapin et al., 2001]. For sake of completion, we therefore decided to add this dimension to our analysis.

We conclude the analysis of each theme by assigning a rating (+, ++, +/−, −, −−) that describes the suitability of a given type system to tackle a given dimension of software evolution. These ratings are then merged in section 5.6 in the form of a table (figure 5).

### 5.1. Temporal Properties (when)

The *when* question addresses temporal properties of software maintenance tasks such as the moment when the change is executed, the change frequency and how different versions of the same software are treated and maintained.

#### 5.1.1. Time of Change

Depending on the programming language and development environment being used, changes can be executed in different phases of the software development cycle. Changes can either be made to the software offline (i.e. while the software is not active) or online (during its execution). The terms *compile-time* and *runtime* are synonymous with *offline* and *online* respectively, and all four terms will be used throughout this paper.

**5.1.1.1. Offline changes** Offline changes are popular because of their ease of implementation: there is no predetermined timeframe in which the change must complete, both testing and verification of the change is possible and a variety of information (such as the source code) is available. Because of this relative ease compared to online changes, offline changes are often used as a preparation stage for online changes: [Vandewoude and Berbers, 2005], [Chiba, 2004], [Tanter et al., 2003]. In doing so, part of the complexity of the online change is shifted offline. This activity, in which additional information known at design time can also be embedded in the code for use at runtime, is often referred to as code instrumentation.

An example of a scenario in which such preparation is often used is *state transfer*. This term refers to the phase (during the runtime replacement of a software entity) in which the internal state of a software entity is transferred (and possibly converted) to the new version. This activity can be prepared offline by adding getter-methods to facilitate the runtime state extraction.

Offline changes in general can obviously be applied to any program, regardless of the language used for its implementation. However, code instrumentation as a preparation for



---

dynamic updates is much more common with statically typed languages. There are two reasons for this phenomenon:

**Information availability:** A static type system provides the user with more information that can be used as a base for analysis or can be included in the code. For instance, in [Vandewoude and Berbers, 2005], code is instrumented with information on corresponding structures between different versions, which is used by the state transfer algorithm. Without the offline instrumentation, such information could never have been derived at runtime. For purely dynamically typed languages, hardly any information is available that can not be easily derived at runtime. Hence, code instrumentation is not used.

**Less powerful reflection:** Purely dynamically typed languages often have a more advanced reflective system which eliminates the use for some forms of code instrumentation (such as the addition of getter methods to extract state information).

*5.1.1.2. Online changes* Generally speaking, static and strong typing hinder online changes such as the removal of a method from a type. To achieve this result, a new type must be created (identical to the old type except for the removed method) and all old objects would need to be converted to their counterpart in the new type. This is clearly more complex than direct type modification such as those allowed by the reflection mechanisms of Smalltalk. This ability to change types after their construction is definitely one of the most powerful advantages dynamically typed languages can offer with respect to dynamic software evolution.

For its lack in flexibility, the type system offers increased safety in return. However, if not all changes are type-safe (and this is often the case in a typical evolution scenario), the type system only gets in the way. In these cases, correctness must be verified using invariant checkers, which can be implemented regardless of the language used. More information on safety is given in section 5.3.4.

#### *5.1.2. Change history*

A change history of a software system refers to the history of all changes that have been made to the software. Static versioning implies that different versions can coexist at compile time, but cannot exist simultaneously at runtime. It is clear that static versioning can be supported for any language. All that is required is a versioning system that keeps the sources of different versions available (such as CVS or Subversion).

Full versioning implies that multiple versions can coexist at runtime. This is a desirable feature since it allows for lazy conversion: the new implementation is used to construct new instances whereas the old instances remain unchanged and are gradually removed from the system after their task has completed. In addition, the ability to have different versions in memory allows for an easy implementation of a rollback functionality. If only one version can be present at a given time, changes are by definition destructive, and rollback is harder to implement.

While full versioning is possible for both dynamically and statically typed languages, it is more naturally supported for the latter, since both versions are considered to be different



types. Nothing prohibits such duplication with purely dynamically typed languages (Smalltalk for instance even allows storage of source code in its classes). However, by doing so, many of the advantages of purely dynamically typed languages are sacrificed. For instance, the ability to change a runtime type through the reflective system can no longer be used because such changes are destructive and as such not compatible with full versioning.

### 5.1.3. *Change Frequency*

The change frequency is inversely proportional to the effort required to achieve a (runtime) change. Since this effort is in general lower for weakly and purely dynamically typed languages, change frequency will be higher for these languages. This process is in part self-sustaining: if changes are more frequent, they tend to be smaller and easier to implement. This in turn decreases the barrier for their use. In practice this is confirmed by iterative development environments for languages such as CLOS and Smalltalk, in which a running program is continuously tested and extended during development.

For statically typed languages, such iterative development environments are virtually non-existent. Live updates, even without any proof of their correctness (such as is the case of Smalltalk) are difficult to achieve, and therefore intentional. Using the terminology of [Buckley et al., 2005], live updates for statically typed languages are almost always *periodical* as opposed to the *continuous* nature of dynamic adaptations in languages without a static type system. Note that technologies such as Java Hotswap [sun, 2002] seem to contradict this statement, since they do allow *on-the-fly* updates with very little effort or preparation. However, these technologies bypass the most difficult problems by strictly limiting their scope to changes of method-bodies.

### 5.1.4. *Anticipation*

Anticipated changes are easier to tackle at runtime than unanticipated changes. The more changes a piece of software anticipates, the better it is armed to deal with evolution later on in its life-cycle. It is clear however that, at the application level, not every possible change can be anticipated.

The main impact of anticipation is that it indicates what portions of the software can easily be changed. The only goal of techniques such as code instrumentation as a preparation for a live update is to increase this portion. The used language and runtime infrastructure have a large impact on anticipation, since all concepts that are modeled as first class entities can be changed at runtime without difficulty. For instance, a message is a first-class entity in Smalltalk and as such, method invocations can be rewired with ease. This would be much harder in Java, where the same concept is not treated as a first-class entity. Both metaobject-protocols and reflection are techniques to anticipate a number of changes at the language level.

In theory, if every concept of the language is fully reified, a language can be constructed that anticipates every possible change and is therefore ideal for dynamically updateable software. Whether the construction of such a language is a realistic undertaking remains subject for debate. However, it is clear that there is a huge difference between different programming languages regarding their support for anticipation. The relation of this support with their



type system is less obvious, and can be explained by looking at the relation between reflective capabilities and the type system as discussed in section 3.4. Languages without a static type system enforce less rules and therefore allow easier implementation of many types of unanticipated changes. As such, purely dynamically typed languages can be considered more anticipative than statically typed languages. The same reasoning can be followed for weakly and strongly typed languages: since weak typing allows the programmer to bypass some of the rules that are dictated by the type system, weakly typed languages are more anticipative than strongly typed languages.

### 5.1.5. Summary

We can summarize the temporal properties as follows:

	Section	Static		Both		Dynamic	
		Strong	Weak	Strong	Weak	Strong	Weak
Time of change	5.1.1						
<i>Offline changes</i>		++	++	++	++	+	+
<i>Online changes</i>		--	--	-	+/-	+	++
Change history	5.1.2	+	++	+	++	+/-	+
Change frequency	5.1.3	--	-	--	-	+	++
Anticipation	5.1.4	-	+	+/-	+	+	++

From this overview, it is clear that static typing hinder the temporal dimensions of dynamic software evolution. This is not surprising since advantages of a static type system are mainly in the area of safety, which is hardly ever considered in the temporal dimension of software evolution. When it comes to speed and flexibility, static type systems only get in the way. However, when *offline* changes are the goal, static type systems provide us with a lot of valuable information that can be used to automate and verify transformations.

## 5.2. Object Of Change (where)

The object of change describes the *where* dimension in the taxonomy of software evolution, as a change is always applied to a specific target. In this section, we assess how the programming language influences the size of the smallest unit of adaptation as well as the impact of the change and its propagation in the rest of the system.

### 5.2.1. Artifact

Software consists of a number of inter-related artifacts such as source code, design models, documentation and test-suites. Dynamic software evolution is exclusively concerned with dynamic artifacts (used at runtime) such as modules, functions and objects. While it is clear that co-evolution of the other artifacts is a desirable feature, the evolution of static artifacts is outside the scope of this paper. Some aspects of dynamic evolution (such as code-instrumentation) should be explicitly excluded from static artifacts, since they do not represent a property of a given version, but are directly related to the evolution process itself. As these artifacts are typically tool-generated, they will automatically evolve when the static artifacts they depend on evolve. They should therefore remain transparent to the software designer.



---

### 5.2.2. Granularity

The degree of granularity defines the minimal size of the unit of replacement. The finer this granularity is, the more difficult it is to ensure consistency. After all, coarse grained components typically contain many tightly-coupled objects that are then replaced in group. Fine grained changes will need to deal with these strong interrelations. However, fine grained changes have the huge advantage that they usually do not require the transfer of complex state. Any state whose structure is not directly changed remains unaffected by a fine-grained change. For instance, a modification in one of the methods of a type would require no modifications whatsoever to the instances of the type. The degree of granularity therefore represents a trade-off between the complexity of state transfer on one hand, and the complexity of maintaining inter-object dependencies consistent on the other hand.

Fine grained changes (within a type) are typically inhibited by statically typed languages, since a large number of changes is forbidden by these languages. With purely dynamically typed languages, there is no static type that an object must correspond to. Hence, a type can easily be changed without the need to create a new type in the process. The powerful reflective mechanisms often found in purely dynamically typed languages proof this fact. While it would be technically feasible to develop an equally powerful reflective API for statically typed languages, the language definition forbids all modifications that could violate type safety, which include most changes to a type.

For instance, it is perfectly possible to implement a Java API that would allow the removal of methods from a Java class at runtime similar to the reflective system found in Smalltalk. If a method is removed and subsequently used, in Smalltalk, a `MessageNotUnderstood` exception is thrown. However, the programmer is aware that such behavior can occur and the language specification dictates that such an event may occur. In Java, such operation breaks the semantics of the programming language, since Java ensures that once the type of an object is known, all its methods can be used. Two options remain: either check dynamically that the removal of the method can never result in an error or dump the guarantee. The former option, if at all possible, would be extremely expensive to do at runtime. The latter option implies removing static type checking from the language, which would result in a Java-syntactic Smalltalk variant.

Statically typed languages can emulate fine-grained changes with coarse grained changes (creating a new type which is identical to the old type except for the fine grained change). However, this is pointless, since the state-preserving features of fine grained changes (their primary advantage) are no longer applicable when this approach is used.

### 5.2.3. Change Impact

The impact of a change applied to a software entity, describes to what extent this change is contained within the boundaries of that entity. Automatically determining whether a change has local or system wide impact is not always trivial. Static typing clearly helps in the localization of affected portions of the system since type information is the main source of impact-analysis. A clear example of the correlation of the type system and its ability to determine change impact can be found in the context of refactoring [Fowler, 1999]. For



statically typed languages (e.g. Java) many refactorings (such as a method rename) can be automated completely. This is impossible for pure dynamically typed languages since they do not provide the means to identify the target of a method invocation or variable access at design time. For instance, the type of the parameters of a method invocation can not be used to identify the correct method. As such, tool support is therefore limited to an exhaustive listing of all possible targets from which the developer must make his selection.

Approaches, such as execution traces [Alur and Dill, 1994], can provide additional information and narrow down the set of possibly affected locations. Nevertheless, such techniques remain limited with respect to static type information. Also note that the information of a static type system is rarely complete. Most statically typed languages (such as Java, C# and Eiffel) rely on dynamic typing to implement concepts as polymorphism and late binding.

#### 5.2.4. *Change Propagation*

Whereas change impact refers to the locality of a given change, the term *change propagation* refers to what extend a certain change requires follow up changes in other encapsulated entities. When a change propagates throughout the system, its total impact on that system increases. For dynamic software evolution, languages or methodologies that limit the propagation of a change are desirable. [Ramil and Lehman, 2000] shows that the impact of a change and the effort to execute it are directly proportional: large sets of affected software entities make the update slower and more complex. The influence of the typing mechanism on change propagation is twofold:

1. Statically typed languages usually prohibit direct modification of types at runtime. Hence, changes often result in the conversion of instances from the old to the new type, sacrificing the identity of the original instance. The change therefore propagates to all structures that contain references to the modified instance. As such, a change with seemingly local impact cascades throughout the system requiring changes in many other entities. For purely dynamically typed languages, direct changes to the type are possible. Such changes often preserve object identity and therefore change propagation is less likely.
2. Coexistence of different versions increases localization of a change since lazy object migration is possible: objects are not actively replaced but are gradually removed from the system when they are no longer used. As mentioned in 5.1.2, this is better supported by static type systems.

The popularity of component-based and aspect-oriented systems for dynamic updating can be explained by their excellent characteristics concerning change impact and change propagation. A component or aspect is considered to be a loosely coupled and self-sufficient entity. As such, changes within a component remain within the boundaries of the component. In most cases, a lookup mechanism separates the identity of a component from an actual instance. Methodologies such as component based software engineering or aspect oriented programming stimulate the separation of concerns. They reduce coupling, limit change propagation and thus increase the ability to perform runtime adaptations [Vandewoude and Berbers, 2004b].



### 5.2.5. Summary

We summarize the *where* theme as follows:

	Section	Static		Both		Dynamic	
		Strong	Weak	Strong	Weak	Strong	Weak
Granularity	5.2.2						
<i>Coarse grained</i>		++	++	++	++	+	+
<i>Fine grained</i>		-	-	-	-	++	++
Impact	5.2.3	++	+	+	+/-	-	--
Change propagation	5.2.4	+/-	+/-	+/-	+/-	+	++

Whereas a static type system had little advantages in the temporal domain, this is no longer true for the dimensions concerning the object of change. Static type systems are an asset when the impact of a change must be accurately estimated. The impact of a change can not be predicted easily without a static type system. On the other hand, static typesystems offer little support for fine grained changes which usually results in increased change propagation vs. purely dynamically typed languages.

### 5.3. System properties (what)

This theme of software evolution refers to a number of system properties of the software system that is being changed and the underlying platform or middleware. It addresses what kind of changes are allowed.

#### 5.3.1. Availability

The goal of dynamic evolution is to minimize downtime. Ideally, the application remains available during the entire adaptation process. In some cases short downtimes are acceptable provided that the new version can continue where the old version was halted (i.e. the state is transferred).

Continuous availability of the entire application is not possible in the presence of state transfer. In order to avoid inconsistencies, the entity under consideration must be placed in an inactive (quiescent) state [Kramer and Magee, 1990] before the actual state is transferred. This causes the application to be partly or completely unavailable (depending on the impact of the change) for the duration of the state transfer. The goal is to minimize both the portion of the application that is inactive and the duration of the update. These two issues are interrelated and are coupled to the granularity: a coarse grained unit of change tends to disable a larger portion of the application for a longer period of time. However, replacing a larger unit is much easier (and often safer), since many tightly coupled constructs are replaced together.

Whenever state transfer is not required, an uninterrupted service can be achieved. Direct type modifications (for example adding a method in a Smalltalk program) do not affect running programs and are fully transparent to the user of the application. For statically typed languages, the lazy update approach can be used (the new implementation is used to construct new entities whereas the old entities are gradually removed from the system after their task has completed): since no active entity is replaced, nothing needs to be deactivated. A major



---

drawback of this technique is that there is no guarantee that the update terminates within a reasonable period of time.

### 5.3.2. *Activeness*

In literature [Oreizy et al., 1999], a distinction is made between *reactive* and *proactive* systems. Whereas changes in a reactive system are driven by an external factor, proactive systems monitor the state of their own execution with sensors and adapt themselves according to a given policy. Since such a policy is always intentionally implemented in advance, proactive systems are by definition anticipated and there is no direct relation with the type system of the language used.

Unanticipated changes are always initiated on request and therefore reactive. The request can be directed to a language runtime environment (e.g. SmallTalk, CLOS, a modified Java VM such as in [Andersson et al., 1998]), a component system (e.g. [Kramer and Magee, 1990], [Vandewoude and Berbers, 2004b]), a meta-layer above the application (e.g. [Redmond and Cahill, 2000]) or even the application itself if it provides a generic interface through which updates can be initiated (such as in [Orso et al., 2002]). Powerful reflective systems ease the implementation of a reactive environment since they provide an implicit maintenance interface to the application. However, limitations of the environment can be overcome with code instrumentation or manual implementation.

### 5.3.3. *Openness*

Software systems are considered to be open if they are specifically built to allow for extensions. With respect to live updates, there is a strong relation between openness and anticipation. Applications are, by definition, closed towards unanticipated changes and open towards anticipated changes. As such, the argumentation given under section 5.1.4 is valid here as well: purely dynamically typed languages are more open due to their powerful reflection mechanism which allow inspection and modification of the language at a very basic level. Additionally, we can say that weakly typed languages will be more open than strongly typed languages.

### 5.3.4. *Safety*

A system features *static safety* if it is able to ensure at compile time that the system will not behave erroneously at runtime. A system provides *dynamic safety* if it contains provisions that prevent or restrict unwanted runtime behavior. The required change support mechanisms are directly influenced by the desired kind and degree of safety.

Static typing provides us with a certain degree of static safety: the system can verify that both the old and the new version are type-safe, and additional analysis can check for type compatibility issues between different versions. It allows, to a certain degree, to verify compatibility at design time or at load time. Without static typing, this is not possible.

At runtime, verification of compatibility can also be performed using techniques as invariant-checking (evaluation of a function that checks the compatibility between the new version and the running system). Such techniques are difficult and time consuming to implement. However,



this is not related to the type system since the implementation of this extra behavior is required anyway. Static type systems only detect type errors, which is clearly insufficient for a typical dynamic adaptation scenario. Many additional conditions must be met for the update to be a success (e.g. access control to initiate the update, consistency checks between versions after a state transfer).

### 5.3.5. Summary

We summarize the *what* theme as follows:

	Section	Static		Both		Dynamic	
		Strong	Weak	Strong	Weak	Strong	Weak
Availability	5.3.1	-	-	+/-	+/-	+	+
Activeness	5.3.2						
<i>Proactive</i>		+/-	+/-	+/-	+/-	+/-	+/-
<i>Reactive</i>		-	+	+/-	+	+	++
Openness	5.3.3	-	+/-	+/-	+	+	++
Safety	5.3.4	+	+/-	+	+/-	-	--

Concerning system support, we see that static typing and dynamic typing are quite complementary: static type systems provide us with safety, whereas a dynamic type system is required to ensure openness and availability. Languages which are both statically and dynamically typed (such as Java) have a clear edge in this area.

## 5.4. Change support (how)

The fourth theme of the evolution-taxonomy describes the change process itself. The actual technique used to accomplish the update is classified according to a number of orthogonal characteristics, some of which are influenced by the programming language used.

### 5.4.1. Degree of automation

In general, it is impossible to achieve full automation when state transfer is involved: it is impossible to correctly identify semantically equivalent structures between different versions of a component since not all semantic information is contained in source code. A typical example is the representation of a triangle. In version  $n$ , three points may be used while version  $n + 1$  uses two edges and an angle. State transfer remains in essence an interactive and at best a semi-automatic process. Nevertheless, tool support can be developed to assist the programmer with this task [Vandewoude and Berbers, 2005]. Such tools are more likely to be successful if they have type information at their disposal<sup>‡</sup>.

When no state transfer is required, a powerful meta-level protocol or reflective system can allow fully automatic execution of declarative change-specifications such as the addition or

<sup>‡</sup>The same statement is valid for other forms of evolution, such as refactoring.



removal of a method. This is an argument in favor of purely dynamically typed languages: it is harder (albeit not impossible) to implement such a system on statically typed languages.

#### 5.4.2. Degree of formality

A change process can be described in various levels of formality. Formal proofs of correctness are extremely useful in the context of dynamic software evolution. After all, an update is worth nothing if not executed correctly. While no such proofs known to the authors are of wide practical applicability, it is not impossible that such formalism could be constructed. Although type information is often used to describe behavior or correctness, other formalisms exist (e.g. graph transformations: see [Mens et al., 2002]) that do not depend on the type system. The relation between a type system and the degree of formality is therefore not absolute. Nevertheless, statically typed languages are generally better suited for formal descriptions.

#### 5.4.3. Change type

There are two types of changes: *structural changes* which are behavior preserving and *behavioral changes*, which by definition change the behavior of the application. Refactorings are a well known example of structural changes and are often used to ease a future behavioral change. Because statically typed languages have more structural information, they are better suited for structural changes than dynamically typed languages (e.g. a large number of refactorings can be completely automated in the presence of a type system). For behavioral changes, dynamically typed languages typically have a slight advantage: they allow for easy modification of method bodies as they support changes of a smaller granularity (see also section 5.2.2).

#### 5.4.4. Summary

We summarize the *how* theme as follows:

	Section	Static		Both		Dynamic	
		Strong	Weak	Strong	Weak	Strong	Weak
Degree of automation	5.4.1	-	-	+/-	+/-	+/-	+/-
Degree of formality	5.4.2	++	+	++	+	+/-	-
Change type	5.4.3						
<i>Structural</i>		+	+	+	+	+/-	+/-
<i>Behavioral</i>		+/-	+/-	+/-	+/-	+	+

In the area of change support, statically typed languages have an overall advantage when the change is complex in nature: both state transfer and complex structural changes are better supported when type information is present. For simpler changes (such as the replacement of a method body), dynamically typed languages are superior as they reduce the generally lower the complexity of these changes.



---

### 5.5. Purpose (why)

Most papers in literature identify four major types of maintenance activities: perfective (enhancing non functional properties), adaptive (implementing extensions), corrective (fixing bugs) and preventive (to ease or avoid later maintenance tasks) [Lientz and Swanson, 1980]. A more extensive taxonomy of maintenance activities based on their purpose is given in [Chapin et al., 2001] in which Chapin et al. identify twelve different types of evolution. In general, the reason why one would attempt a runtime change to a running program is independent from the technology that is used to achieve this goal. Nevertheless, the better a technology supports runtime software evolution, the more likely it is that such technology will actually be used.

In statically typed languages, performing a runtime change is generally quite difficult, since one must uphold guarantees of the language that no type errors can occur in the process of a change. While certainly not impossible, such an attempt is non-trivial and requires a lot of preparation. It is very unlikely that one would take the effort to do so unless it can not be avoided. Therefore, it is common to make some preventive changes offline which are intended to ease later evolution (such as the addition of getter methods discussed above).

For some purely dynamically typed languages however, certain forms of dynamic modification can be executed with relative ease. The `become` operator in Smalltalk allows switching two objects, is easy to use and has global impact on the active image. In addition, interactive development environments (such as the Smalltalk browser, but also some Prolog development environments) allow modifications so easily that runtime changes are truly incorporated in the development process: programs are constructed through incremental change. This process is essentially part of software development and differs from true live updates since many important problems (such as state transfer) are not addressed. As such, since such iterative and prototype driven development offers less safety, corrective bugs are rather common for purely dynamically typed languages.

Although some impact from the type system on the purpose of evolution can therefore be found, this impact is indirect at best and is more related to the easyness with which the dynamic updates can be executed. We therefore feel that the relation between the type system and the purpose of evolution is not sufficient to warrant inclusion in our final overview table.

### 5.6. Putting it all together

The table of figure 5 summarizes the findings of this paper by merging the results from the different sections. The overview distinguishes between six different typing combinations ranging from static/strong to dynamic/weak and confirms the expectations from section 3.4. It demonstrates that weaker typing corresponds with increased power to carry out the changes but also that this flexibility comes at a cost: reduced impact estimation, lower safety and more difficult formalizations.

This table can be used in combination with the language-overview from the section 3.4 (see figure 3) to assist a developer with the selection of a programming language. The table also shows that every type system (be it weak/strong/static/dynamic) has its strengths and



Theme	Dimension	Section	Static		Both		Dynamic	
			Strong	Weak	Strong	Weak	Strong	Weak
Temporal (when)	Time of change	5.1.1						
	<i>Offline changes</i>		++	++	++	++	+	+
	<i>Online changes</i>		--	--	-	+/-	+	++
	Change history	5.1.2	+	++	+	++	+/-	+
Object of change (where)	Change frequency	5.1.3	--	-	--	-	+	++
	Anticipation	5.1.4	-	-	+/-	+/-	+	+
	Granularity	5.2.2						
	<i>Coarse grained</i>		++	++	++	++	+	+
System properties (what)	<i>Fine grained</i>		-	-	-	-	++	++
	Impact	5.2.3	++	+	+	+/-	-	--
	Change propagation	5.2.4	+/-	+/-	+/-	+/-	+	++
	Availability	5.3.1	-	-	+/-	+/-	+	+
Change support (how)	Activeness	5.3.2						
	<i>Proactive</i>		+/-	+/-	+/-	+/-	+/-	+/-
	<i>Reactive</i>		-	+	+/-	+	+	++
	Openness	5.3.3	-	+/-	+/-	+	+	++
Change support (how)	Safety	5.3.4	+	+/-	+	+/-	-	--
	Degree of automation	5.4.1	-	-	+/-	+/-	+/-	+/-
	Degree of formality	5.4.2	++	+	++	+	+/-	-
	Change type	5.4.3						
	<i>Structural</i>		+	+	+	+	+/-	+/-
	<i>Behavioral</i>		+/-	+/-	+/-	+/-	+	+

Figure 5. Type impact on software evolution

weaknesses. Therefore, it is not surprising that no language exists that is ideally suited for every aspect of software evolution.

The research described in this paper can also be considered as a real-life test of the evolution taxonomy itself. Overall, the taxonomy performed well. It succeeded in providing us with a framework in which the different topics of discussion can be situated. Nevertheless, it seems clear that the taxonomy can be further improved with the addition of an additional dimension: *language properties*. As clearly shown in this paper, the programming language in which a piece of software is developed has a big impact on its evolution characteristics. The newly added dimension fits nicely in the *system properties*-theme of the taxonomy.

## 6. Example

We illustrate the impact of typing on evolution with an example which is rather common in practical software development: exception handling. Imagine a hierarchy of exceptions, and an application consisting of a number of classes that throw these exceptions. For instance, in figure 6, we consider an application that processes some complex task. This task includes the printing of a file, during which a number of exceptions can be thrown. The relevant part of the original application is shown above the dotted line in figure 6. After a while, support for a networkprinter is added to the system. Being a printer, `NetworkPrinter` evidently inherits

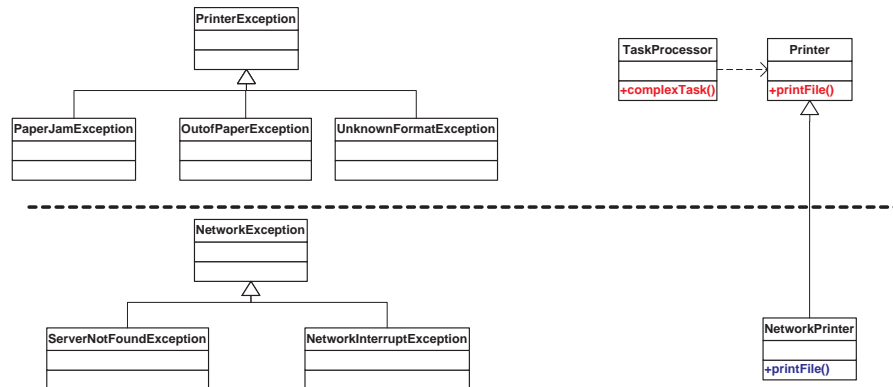


Figure 6. Influence of checked exceptions on evolution. The original application is shown above and the extensions below the dotted line. While only one method is changed (`printFile`), the consequences ripple through to all parentclasses (in this case `Printer`) and all classes that call the changed method (such as the `complexTask` method of `TaskProcessor`).

from the `Printer` class. However, a number of network-related exceptions are introduced as well.

Although ideally the introduction of a new subclass should not induce changes in the parent, this is not the case when checked exceptions are present. The `printFile` method must declare the new exceptions as well. In addition, all methods that call this functionality (such as the `complexTask` method of `TaskProcessor`) must be changed as well (either to handle the exception, or to declare it in the exceptionclause of the method). In figure 6, the intended change is shown in blue, whereas all unwanted changes are marked in red. For purely dynamically typed languages, this is not a problem, as these languages do not declare possible exceptions in the method signature.

This small example illustrates many of the results from this paper. First and foremost, it is a clear illustration of the dimension *change propagation* for which purely dynamically typed languages are declared superior: only for statically typed languages, the programmer is required to change a large number of other classes in the application. Evidently, this is also reflected in the *change impact* dimension: the impact of the change is much more clearly known for statically typed languages as the compiler identifies all possible change locations. Another possible view on the same issue is that dynamically typed languages better anticipate this kind of change (reflected in their higher ranking for the *anticipation* dimension). Checked exceptions also improve the *safety* of the program, since it is not possible to encounter an unanticipated exception at runtime. After all, the programmer is forced to take a decision for every checked exception, so he can be reasonably sure that all checked exceptions are handled properly. All these findings are reflected in figure 5.

In practice, many statically typed languages try to overcome the problems of checked exceptions by allowing *unchecked exceptions* to be defined which do not need to be declared.



---

Evidently, this solution sacrifices the advantages of checked exceptions in the progress. Only very recently, new language constructs such as anchored exceptions succeed in combining the ease of use of unchecked exceptions with the safety of checked exceptions, by declaring the exceptions of a method relative to that of other methods [van Dooren and Steegmans, 2005].

## 7. Related and future work

So far, the relation between programming language features and dynamic evolution has not been systematically investigated. Overviews and comparisons of different systems for dynamic adaptation do exist however, and can be found in [Hicks, 2001], [Segal and Frieder, 1993], [Vandewoude and Berbers, 2002]. While no programming language is specifically designed with dynamic evolution in mind, Hicks modified the Popcorn programming language in order to provide better support for online evolution [Hicks, 2001]. To achieve this goal, more language constructs were reified into a first-class entity so that they could be inspected and/or modified.

This paper primarily dealt with the impact of the type system on dynamic software evolution. Future work includes an analysis of other language properties that directly or indirectly influence dynamic adaptations. The majority of programming languages uses lexical scoping since this is more intuitive than dynamic scoping. However, [Costanza, 2003] shows that dynamic scoping is more flexible and possibly better suited for evolution. Another second important characteristic is the programming paradigm: functional languages do not require state transfer, prototype based languages are flexible and have many possibilities for evolution as they have no classes. Here too, flexibility comes at the cost of reduced encapsulation and security. And while aspect oriented programming aims to provide clear separation of concerns, [Tourwe et al., 2003] shows that resulting code is not necessarily loosely coupled.

The execution system of a program has impact on evolution properties as well. Interpreted languages are easier to change at runtime than compiled languages. Languages which are compiled to intermediary code (such as Java or C#) often use class loading techniques to overcome certain issues. While successful to some extent, they are clearly more complex than their interpreted counterparts [Chiba, 2004], [Tanter et al., 2003].

A final language feature that has possible implications on evolution is concurrency. Two major techniques exist to avoid data clashes in multi-threaded systems: the shared data and the actor-model approach. In [Agha et al., 1997], the authors claim that the actor-model approach is better suited for evolution since concurrency functionality is grouped in one place.

## 8. Conclusion

Live updates (evolving a program without shutting it down) is a complex and hot topic in software research. So far, no programming language has been designed with dynamic software evolution in mind. Therefore, current techniques all require significant additional effort from the designer and the programmer of the software to compensate for the lack of language support.



The primary contribution of this paper is that it thoroughly investigates the relation between a programming language's type system and its suitability for dynamic adaptation. A existing taxonomy of software evolution [Buckley et al., 2005] is used as the starting point for our analysis. Our findings resulted in a concise table that clearly summarizes the relation between a type system and evolution. This table shows that the more flexible the type system, the more power is available to carry out actual changes. It also shows that these powers come at a cost: reduced security. This table can either be used as a starting point for language designers that want to develop a language for dynamic adaptation, or by application developers to select a suitable language according to their needs. Finally, this study suggests an additional dimension for inclusion in the taxonomy: *language features*.

## REFERENCES

- [sun, 2002]. (2002). The java hotspot virtual machine: A technical white paper. Technical report, Sun Microsystems.
- [Abelson and Sussman, 1985]. Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.
- [Agha et al., 1997]. Agha, G., Mason, I. A., Smith, S. F., and Talcott, C. L. (1997). A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72.
- [Alanko, 2004]. Alanko, L. E. (2004). Types and reflections. Master's thesis, University of helsinki.
- [Alur and Dill, 1994]. Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.
- [Andersson et al., 1998]. Andersson, J., Comstedt, M., and Ritzau, T. (1998). Run-Time support for dynamic Java Architectures. In *ECOOP'98 Workshop on Object-Oriented Software Architectures*, Brussels, Belgium.
- [Brodsky et al., 2001]. Brodsky, A., Brodsky, D., Chan, I., Coady, Y., Gudmundson, S., Pomkoski, J., and Ong, J. S. (2001). Coping with evolution: Aspects vs. aspirin. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*.
- [Buckley et al., 2005]. Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5).
- [Cartwright and Fagan, 1991]. Cartwright, R. and Fagan, M. (1991). Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292.
- [Chapin et al., 2001]. Chapin, N., Hale, J. E., Kham, K. M., Ramil, J. F., and Tan, W.-G. (2001). Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30.
- [Chiba, 2004]. Chiba, S. (2004). Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1).
- [Costanza, 2003]. Costanza, P. (2003). Dynamically scoped functions as the essence of aop. In *Workshop on Object-Oriented Language Engineering for the Post-Java Era*.
- [Ebraert and Tourwe, 2004]. Ebraert, P. and Tourwe, T. (2004). A reflective approach to dynamic software evolution. In Cazolla, W., editor, *In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04) in conjunction with the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway.
- [Finkel, 1996]. Finkel, R. A. (1996). *Advanced Programming Language Design*. Addison-Wesley.
- [Fowler, 1999]. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- [Hicks, 2001]. Hicks, M. (2001). *Dynamic Software Updating*. PhD thesis, University of Pennsylvania.
- [Kramer and Magee, 1990]. Kramer, J. and Magee, J. (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306.
- [Lientz and Swanson, 1980]. Lientz, B. and Swanson, E. (1980). *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley.
- [MacLennan, 1986]. MacLennan, B. J. (1986). *Principles of programming languages: Design, Evaluation, and implementation*. Ted Buchholz, second edition edition.
- [Maes, 1987]. Maes, P. (1987). *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel.



- 
- [Mens et al., 2002]. Mens, T., Demeyer, S., and Janssens, D. (2002). Formalising behaviour preserving program transformations. In *Proceedings of the First International Conference on Graph Transformation*, Barcelona, Spain.
- [Mens and Wermelinger, 2002]. Mens, T. and Wermelinger, M. (2002). Separation of concerns for software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):311–315.
- [Opdyke, 1992]. Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Phd thesis, University of Illinois at Urbana Champaign.
- [Oreizy et al., 1999]. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, pages 54–62.
- [Orso et al., 2002]. Orso, A., Rao, A., and Harrold, M. (2002). A technique for dynamic updating of java software. Technical Report GIT-CC-02-24, College of Computing, Georgia Institute of Technology.
- [Ramil and Lehman, 2000]. Ramil, J. F. and Lehman, M. M. (2000). Metrics of software evolution as effort predictors - a case study. In *Proc. Int. Conf. Software Maintenance*, pages 163–172.
- [Rausch, 2000]. Rausch, A. (2000). Software evolution in componentware - a practical approach. In *Proc. of the Australian Software Engineering Conference*.
- [Redmond and Cahill, 2000]. Redmond, B. and Cahill, V. (2000). Iguana/j: Towards a dynamic and efficient reflective architecture for java. *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*.
- [Schmidt, 1994]. Schmidt, D. A. (1994). *The structure of Typed Programming Languages*. MIT Press.
- [Scott, 2000]. Scott, M. L. (2000). *Programming Language Pragmatics*. Morgan Kaufmann Publishers.
- [Segal and Frieder, 1993]. Segal, M. E. and Frieder, O. (1993). On-the-fly program modification. *IEEE Software*, 10(2):53–65.
- [Sethi, 1996]. Sethi, R. (1996). *Programming Languages: Concepts and Constructs*. Addison-Wesley, second edition.
- [Tanter et al., 2003]. Tanter, E., Noyé, J., Caromel, D., and Cointe, P. (2003). Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R. and Steele, Jr., G. L., editors, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, USA. ACM Press. ACM SIGPLAN Notices, 38(11).
- [Tourwe et al., 2003]. Tourwe, T., Brichau, J., and Gybels, K. (2003). On the existence of the aosd-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*.
- [van Dooren and Steegmans, 2005]. van Dooren, M. and Steegmans, E. (2005). Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. ACM Press.
- [Vandewoude and Berbers, 2002]. Vandewoude, Y. and Berbers, Y. (2002). Overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proceedings of SERP02*, pages 521–527.
- [Vandewoude and Berbers, 2004a]. Vandewoude, Y. and Berbers, Y. (2004a). Fresco: Flexible and reliable evolution system for components. In *Electronic Notes in Theoretical Computer Science*.
- [Vandewoude and Berbers, 2004b]. Vandewoude, Y. and Berbers, Y. (2004b). Supporting runtime evolution in seescoa. *Journal of Integrated Design & Process Science: Transactions of the SDPS*, 8(1):77–89.
- [Vandewoude and Berbers, 2005]. Vandewoude, Y. and Berbers, Y. (2005). Deepcompare: Static analysis for runtime software evolution. Technical Report CW405, KULeuven, Belgium.
- [Wuyts, 2001]. Wuyts, R. (2001). *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel.
-