**============================================================================**

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

**4.2   Language Style**

The second category with a major influence on the collection design is the language for which the library is designed.

The *language style* in terms of being procedural, object-oriented, or functional leads to different designs in shape and structure of collections as well as the provided sets of operations on them and the operation naming.

The language’s stance on *typing* and its support for type parameters shapes libraries in various ways. In dynamically-typed languages, we see the need for explicit support of collections for primitive types, e.g., to store numeric data efficiently. This includes sequences for numeric elements (e.g. Int32Array in ECMAScript) or primitive maps or sets (e.g. IntMap in Scala). Typed languages often use some form of generics, type parameters, or templates, which help to reduce a proliferation of collections for specific data types.

In addition to typing, the *reuse* mechanisms offered by a language have an important impact on library structure. For example, single inheritance can lead to designs with undesirable properties as seen in the Smalltalk-80 collection library [[12](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XCook:1992:ISS)]. While some of these issue can be worked around, other reuse mechanisms such as traits, might result in designs that have benefits [[3](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XBlack:2003:ATS)].

In some case, languages evolved to facilitate desired collection library designs [[9](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XChakravarty:2005:ATC), [19](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XGarcia:2007)]. In other cases, the library needed to evolve and use the available language mechanisms more effectively to improve maintainability and code reuse [[30](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XOdersky:2009)].

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

**FROM 4.3**

***Multi-Threading Support.*** To support multi-threaded applications, libraries can include various collection types. The simplest solution to ensure correctness is to provide *synchronization wrappers* as done by Java and C#. However, this approach is rarely efficient and often lacks support for performing multiple operations safely together. These problems are typically addressed by specialized *concurrent* collections, which are optimized for specific use cases, leading to a proliferation of collection types. For instance a queue for a single producer but multiple consumers can often be implemented more efficiently than a queue for multiple producers.

For efficient bulk operations, languages such as Scala provide *parallel* collections, which execute operations in parallel. Similar to sorting, where some collections do it intrinsically, parallel execution is often provided via operations external to the collection (cf. section 4.4).

**============================================================================**

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

**5   Collection Usage**

As seen in the previous section, collection libraries differ widely in the number of collection types they provide and the properties these collections can offer. Especially Java provides a multitude of different collection types and still, the wider community felt the need to provide and maintain libraries for additional collections. One important question to guide the design could therefore be: which collection types are widely used and are likely going to be needed for exploratory programming tasks?

To answer the question of which collection types are widely used, Costa et al. [[14](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XCosta:2017:ESU)] studied a GitHub corpus of Java projects [[1](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#X6624029)]. They found that most instantiation sites for collections create ArrayList objects. From all sites analyzed, 47% used the standard Java ArrayList. Overall, about 56% used some kind of list. Maps where used by about 28%, where the great majority uses Java’s HashMap, which results in a total of about 23% of all allocations. About 15% all instantiation sites were for some set type. Again, the large majority was for Java’s HashSet with about 10% of all allocations.

The only other study on collection usage we are aware of was included in work by Bergel et al. [[2](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XBergel:2018), sec. 9.2]. They observed that OrderedCollection (similar to ArrayList) and Dictionary (a map) are the most frequently used collections in some larger Pharo Smalltalk projects. While the study is less comprehensive than the Java one, it confirms the general trend. It also considers Smalltalk arrays and finds that they are used slightly more often than Dictionary.

**============================================================================**

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

**6.1   Identifying a Point in the Design Space**

***Types of Collections.*** The first and perhaps most important choice seems to be the selection of the desired set of collection types. In section 4.1, we identified six groups. However, considering the usage in actual applications (cf. section 5), only sequences, sets, and maps seem to be widely used. To avoid unnecessary decisions, we propose to use only a minimal set of collection types based on the most widely used ones, and to provide relevant functionality as part of these collections. Furthermore, we suggest to design collection properties in a way that one can easily opt into properties or opt out from them. However, this should be designed together with tool support to avoid losing relevant feedback from the environment. For instance, if a list is used as a stack, the environment should pick up on it, and adapt code completion accordingly.

It can be debated whether to include sets or not. Sets can be easily emulated with maps or with set operations on sequences. Since the use and semantics of sets is distinct enough from sequences and maps, and the various possible designs of emulating sets have drawbacks, we would include them directly, even though there are a number of languages that do not do so.

Similarly, one can argue that maps are merely lists of pairs or that all sequences should be associative arrays. We agree that it can be beneficial to threat them as being polymorphic, and Lua is a great example that shows having only associative arrays (cf. section 3) is practical. However, we consider the usage of maps and sequences as sufficiently distinct to warrant the distinction between them.

Thus, we propose to include *sequences*, *maps*, and *sets*.

***Language Style.*** Generally, we consider the language style as a given from the host language and thus, it is not part of this discussion. However, aspects such as typing can have a major influence on the design. For dynamic languages, we observed the inclusion of specific collection types for primitive data. Since this increases the number of explicit choices one has to consider, we would argue that it is better to rely on optimizations at the implementation level for use cases where performance is the main driver (cf. section 7).

For typed languages, there seems to be a high degree of exposed complexity for users. For example in Scala, the collection library is designed so that operations on collections produce output collections with the same type as the input collection [[30](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XOdersky:2009)]. This leads to an exposure of highly complex type signatures to the user. Scala choses to mitigate this by including simplified type signatures in its documentation. Thus, we advice collection designers to consider this issue and ensure that tooling and documentation hide such accidental complexity.

***Properties.*** Because of the various properties collections might want to support, it seems best to decide on the most flexible default case and additionally consider mechanisms to opt into or out from certain properties.

A good example is whether to offer fixed or variable sized collections. Variable sized ones seem to be the most flexible solution. However, to facilitate the use case and optimizations for fixed-size collections, it is useful to offer for instance constructors that allow creating a collection with the desired size and default value. Special purpose collections such as bounded queues could then be provided as external libraries.

Similarly, maintaining insertion order for all collections seems to be a choice that guarantees deterministic behavior and thus is often preferable. Other orderings could be offered with operations or iteration constructs. This also means that sorting is arguably something one wants to opt into, for instance by requesting a collection to be sorted or using operations that maintain sorting explicitly.

On the other hand, the choice between having mutable and immutable collections is likely tightly bound to the language style. In the interest of maintaining a minimal set of collections only, deciding on either mutable or immutable seems to be preferable and avoid confusion and duplication. Similarly, we would relegate read-only wrappers or copy-on-write collections to external libraries.

Thread-safety is desirable for languages that support sharing collections between multiple threads. As discussed in section 7, we think this can be provided implicitly without drawbacks. However, to provide atomicity of the right granularity, collections need high-level operations such as computeIfPresent or putIfAbsent, which typically check some condition on the collection, potentially perform a user-specified operation, and then modify the collection.

To keep the set of collections minimal, parallel execution of bulk operations is best introduced by orthogonal means. This could mean as part of operations for internal traversal or mechanisms for external traversal, possibly using streams.

Whether to support strict or lazy operations seems to be a question of language style and the alternative seems to be a candidate for external libraries.

Weakly referencing collections have many important applications, but remain special purpose, thus, should be part of an external library as well.

Identity-based maps and sets are ideally realized by parameterizing the collection. Ideally, it has defaults appropriate for the language, which can be easily customized.

***Operation Design.*** For a large part, we assume that language style and type dictate a certain operation style. However, operations on collections might be especially easy to access for instance with good code completion. Thus, supporting a wide range of internal iteration operations seems useful and can support complex queries. Concepts such as loops or streams for external iteration can still be beneficial, too. We also argue that these operations are easier to discover than operations hidden in some complex hierarchy of special purpose collection classes. Thus, we consider a small set of collection types with a large number of operations as a choice beneficial for exploratory programming. Furthermore, ensuring a high degree of polymorphism between collections seems especially desirable in the exploratory stage, because it allows switching between collections or generalizing code without accidental complexity and technical issues distracting form the problem to be solved.

***Algorithms, Data Structures, and Implementation Choices.*** Having to pick any specific algorithm or internal representation of a collection in the exploratory programming stage seems to be solely a distraction. While there can be important performance difference, we would argue that it is better to forgo perfect performance and instead expect the underlying implementation to be *sufficiently smart*. As argued in section 7, it seems feasible and practical to make this tradeoff.

**============================================================================**

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

**FROM 6.2**

Another argument supporting our choice is that it is desirable to separate the optimization concerns from the algorithms. Such separation is a common goal for high-performance computing languages, such as the partitioned global address space languages [[16](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XPGASSurvey)], to make concerns such as the memory layout and access patterns for matrixes and arrays secondary issues, which can be handled as annotations or properties of data structures. This separation allows domain experts to build algorithms, which then later can be optimized by performance experts that ideally only need to add some annotations to ensure an efficient execution.

More generally, we believe that performance-driven choices are rarely possible in an informed manner during exploratory programming. Having not yet fully understood the problem, one would likely mispredict the distribution of operations at run time. Therefore, specialized classes are unlikely to be beneficial.

**============================================================================**

[**https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/**](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/)

Xu [[40](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XXu2013)] and Costa and Andrzejak [[13](https://stefan-marr.de/papers/px-marr-daloze-few-versatile-vs-many-specialized-collections/#XCosta:2018:CS)] go beyond simply avoiding boxing. They show that collections can be adapted further to take concrete usage in terms of used operations into account. For example, if a contains() operation is used frequently on large lists, it can be beneficial to change its implementation to include a hash table to speed up the lookup.

**============================================================================**

[Laurence Tratt: Storage strategies for collections in dynamically typed languages](https://tratt.net/laurie/research/pubs/html/bolz_diekmann_tratt__storage_strategies_for_collections_in_dynamically_typed_languages/)

At 1. Introduction

Implemented naively, dynamically typed languages tend to have poor performance relative to statically typed languages [[35](https://tratt.net/laurie/research/pubs/html/bolz_diekmann_tratt__storage_strategies_for_collections_in_dynamically_typed_languages/#Xtratt__dyamically_typed_languages)]. The flexibility and dynamism of dynamically typed languages frustrates most traditional static optimisations. Just-In-Time (JIT) compilers defer optimisations until run-time

**============================================================================**

[Laurence Tratt: Storage strategies for collections in dynamically typed languages](https://tratt.net/laurie/research/pubs/html/bolz_diekmann_tratt__storage_strategies_for_collections_in_dynamically_typed_languages/)

Second last paragraph from the end of 1. Introduction.

PyPy’s storage strategies are approximately 1500 LoC in total.

**============================================================================**