A Dependency Flip Book

Dirk Engel, info@engel-internet.de*, October 14, 2023*
https://github.com/dirkengel/

Introduction

Just as I was thinking about different types of SW dependencies that could be distinguished, a new episode of 'SW Architektur im Stream' [Wolff23] on the topic was released, where Eberhard Wolff discusses the article “*How an Ancient Philosophy Problem Explains Software Dependence”* by Jimmy Koppel [Koppel22].

My initial idea was to differentiate types of dependency according to time (Coding, Compiling, Linking, Runtime) when they become perceptible. Koppel’s approach is going further and distinguishes between three dimensions. While his idea is compelling, the explanation is somewhat abstract and theoretical. To get a better intuitive understanding of the proposed idea, I admittedly had to watch the video and scroll through the article more than once. Therefore, I wondered if it would be possible to map my original idea onto the three-dimensional approach and find a simple, low-threshold way to visualize dependencies via diagram. The result is this flip book.

Three Dimensions

Koppel distinguishes three dimensions: *Semantics*, *Properties*, *Permitted Changes*. I dare say that my identified points in time at which the dependency becomes noticeable are largely consistent with the **Semantics** factor. Since we usually have tools such as generator, compiler, linker, interpreter and processing unit that do the formal semantic checks for us, I differentiate between the following phases:

1. Model Change (Code Generation)
2. Code Change
3. Compilation
4. Linking
5. Runtime Availability
6. Runtime Conditions

To represent the **Property** factor, I chose the quality attributes defined by ISO 25040 [ISO11]:

1. Functional Suitability: Explicitly and deliberately selected and actively managed dependencies to achieve desired/specified functionality. Followed by the quality attributes on which software depends, whether desired or not. If ignored, it often results in negative consequences:
2. Reliability
3. Security
4. Performance
5. Compatibility
6. Portability
7. Maintainability
8. Usability (not UIX but in terms of API)

Finally, Koppel's third and final factor, **Permitted Changes**, adds the restriction that a change should not add new (unwanted) dependencies. Another example of such a restriction is avoiding cyclic build dependencies.

Notation and Visualization

The **x-axis** shows the T*ime of First Consequences*, which corresponds to the first time at which a dependency becomes apparent by checking *Semantics*. For me, an earlier consequence means a stronger dependency. Although, for example, a C*ode Change* requires a re-*Compilation*, only the first (stronger) consequence is plotted.

To make the *Properties*, which are sometimes overshadowed by the plain functionality, visible, they are illustrated along the **y-axis**. The more quality attributes are affected, the deeper the dependency.

The **z-axis** is for the specific dependencies between modules, written as A→B, which means module A depends on module B. The undesirable case of cyclic dependence is highlighted by making it visible in the same diagram as A←B. Sometimes multiple individual dependencies can be expressed by an equivalence class A→$[A]_R$, which represents a set of modules that share a special property with module A expressed via relation R. If n individual dependencies of a module can be reduced to a single generic equivalence class dependency, it is preferable.

Along the x-Axis

X1 If it's possible to describe the mechanics at a meta level, using code generation is more efficient than manually writing code with recurring common parts. The **Model Change** step occurs before any additional manual code changes that may be necessary to integrate the generated code. Regarding the dependency depth along the *y-axis*, refer to X2 *Code Change* as it is still a code change.

X2 Any **Code Changes** are made to provide or improve functionality. When writing code, you must follow paradigms and rules to produce reliable, secure, and performant code. Thoughtful use of resources supports co-existence (compatibility). Modularity facilitates portability and maintainability. Usability can be achieved by adhering to common API and error handling patterns.

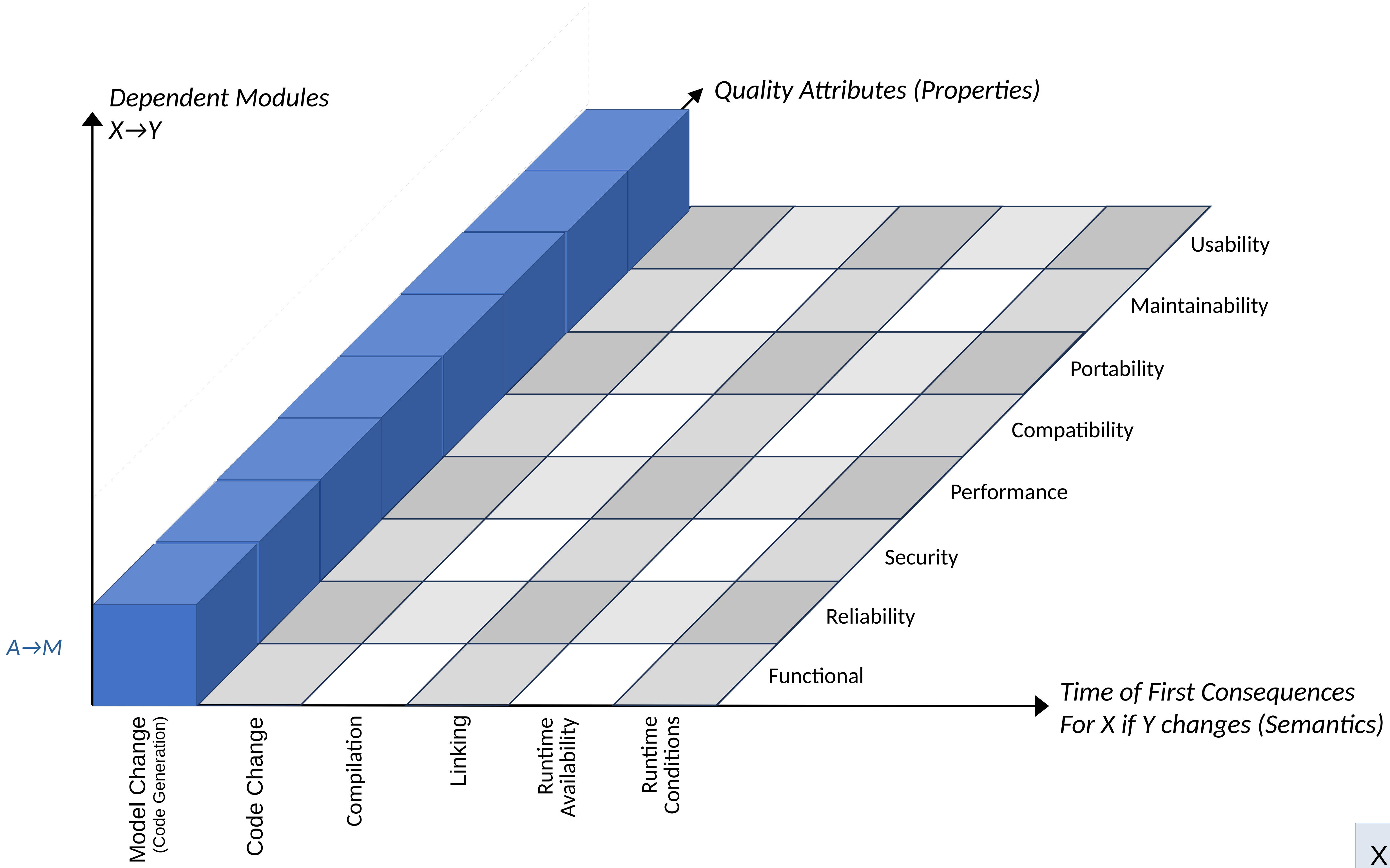
X3 When you need to compile against code you depend on, you may directly experience issues related to unreliability, insecurity, and poor performance introduced to your shared **Compilation** unit. The other code should also be considerate of resource usage to ensure co-existence (compatibility). Since all code contributes to one unit, all modules are closely interconnected in terms of portability and maintainability. If you reuse classes as parameters for your own functions, you also rely on their usability, for instance.

X4 When **Linking** another library, you still depend on its functionality and quality; it's just that the code is already compiled. If you do not have to recompile when the other module changes, your interfaces obviously remain stable, so there is no impact on usability.

X5 The **Runtime Availability** means that the program functionality is made available at runtime. This can be in the form of a dynamic library or a server connection. Since a dynamic library runs in the same process context, it behaves similarly to X4 *Linking* in terms of reliability, security, performance, and compatibility. The same considerations apply to portability and maintainability, as the dynamic library must be built with a compatible build chain and settings, and it also needs to be considered when replacing dependent modules.
In the case of a server the connection has to be established. Functionality, reliability, security, and performance are influenced by the interface and protocol, as well as the quality of the connection. However, the server is a separate instance in terms of compatibility, portability, and maintainability

X6 Now that all modules are assembled to provide the desired functionality, their reliability (availability), security, performance, and compatibility (co-existence) still depend on cooperativeness and the **Runtime Conditions**. Structural attributes such as portability, maintainability, and API usability are addressed in earlier phases.

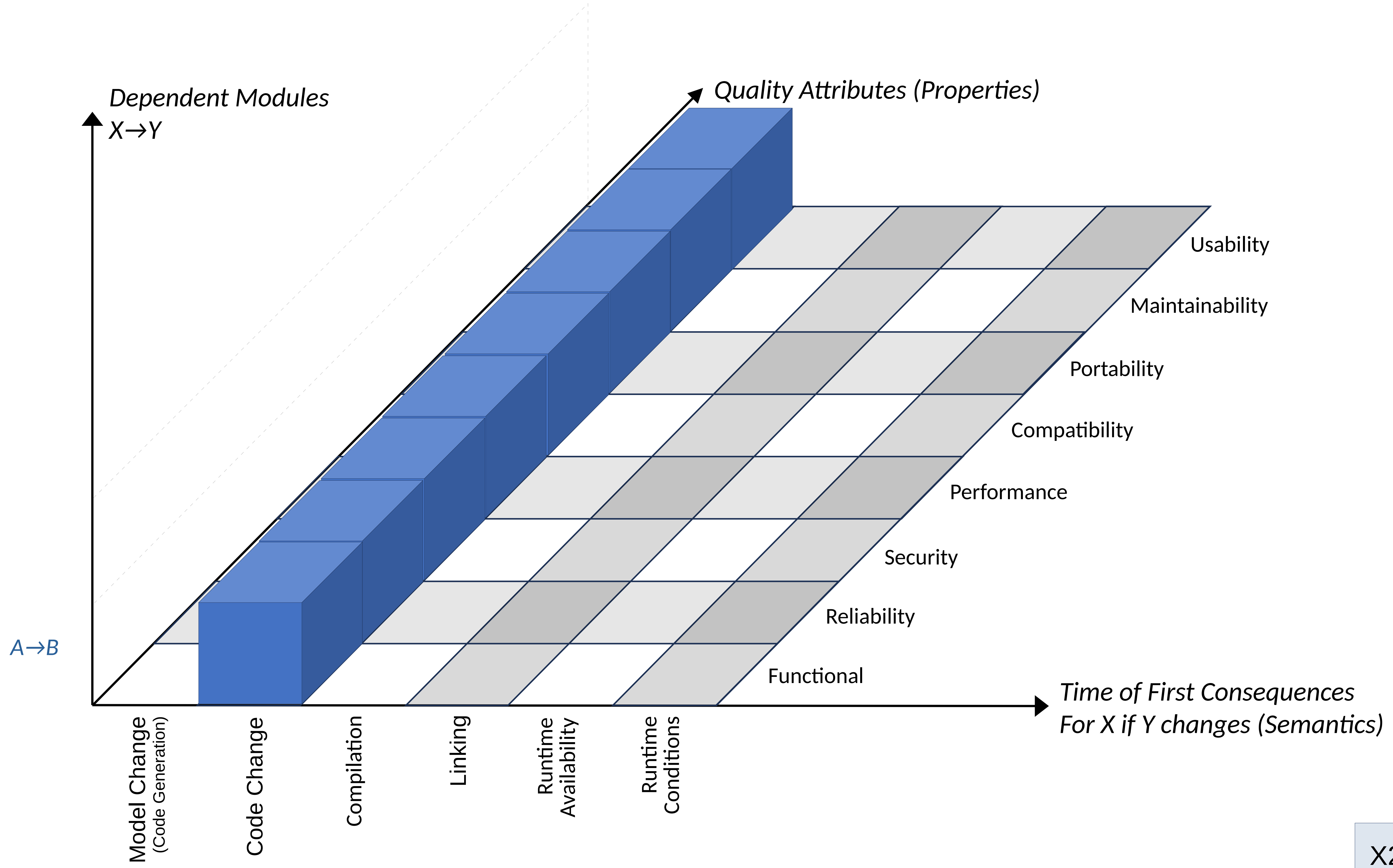
Module A depends on model M that is used to generate source code.



X1

Module A depends on module B in such a way that the source code of A has to be changed when B is changed.

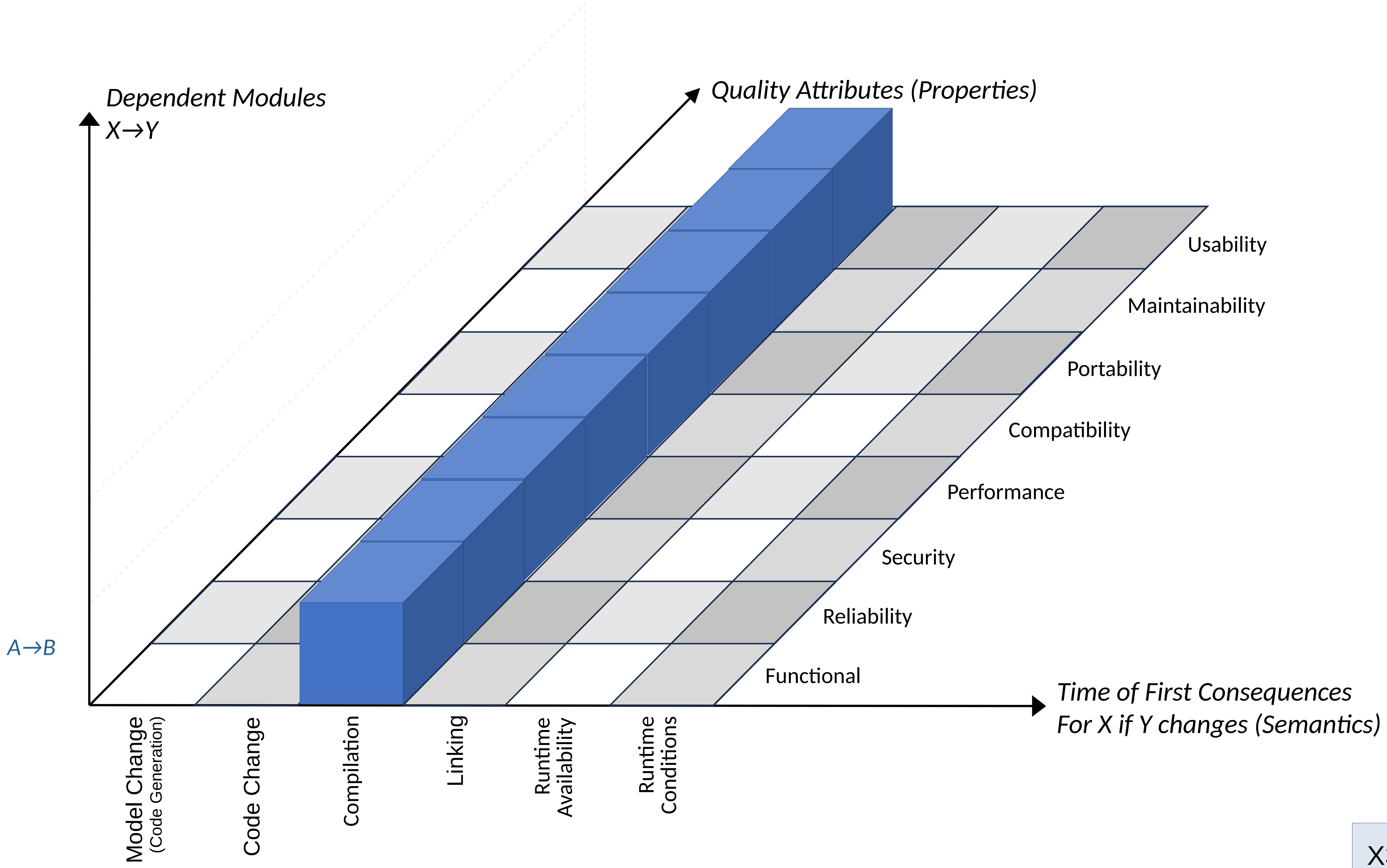
Depending on the definition of what constitutes a module, it is worth questioning whether this division really makes sense or whether A/B is better considered as a single unified component (high cohesion).



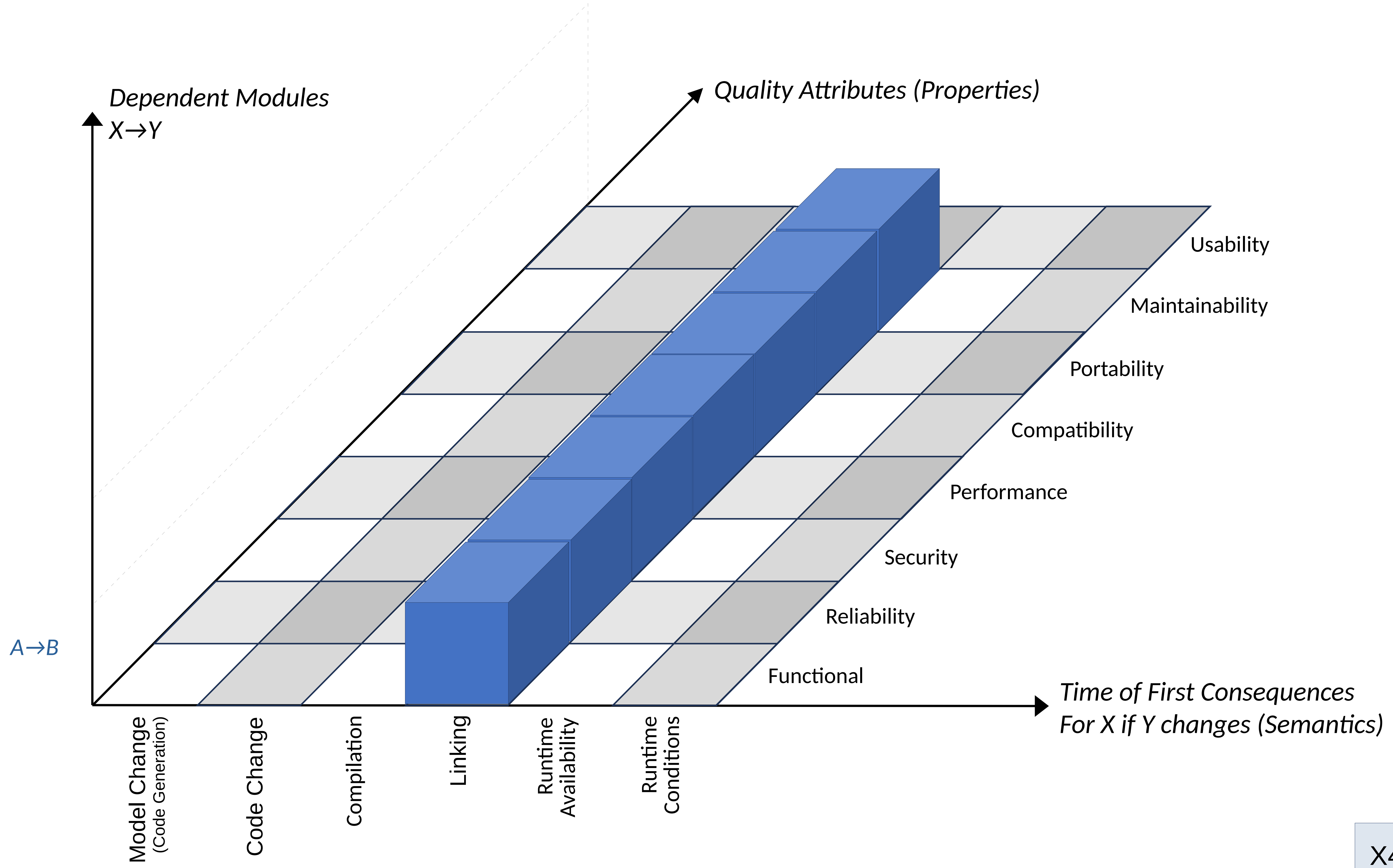
X2

Module A depends on module B and has to be recompiled when B changes.

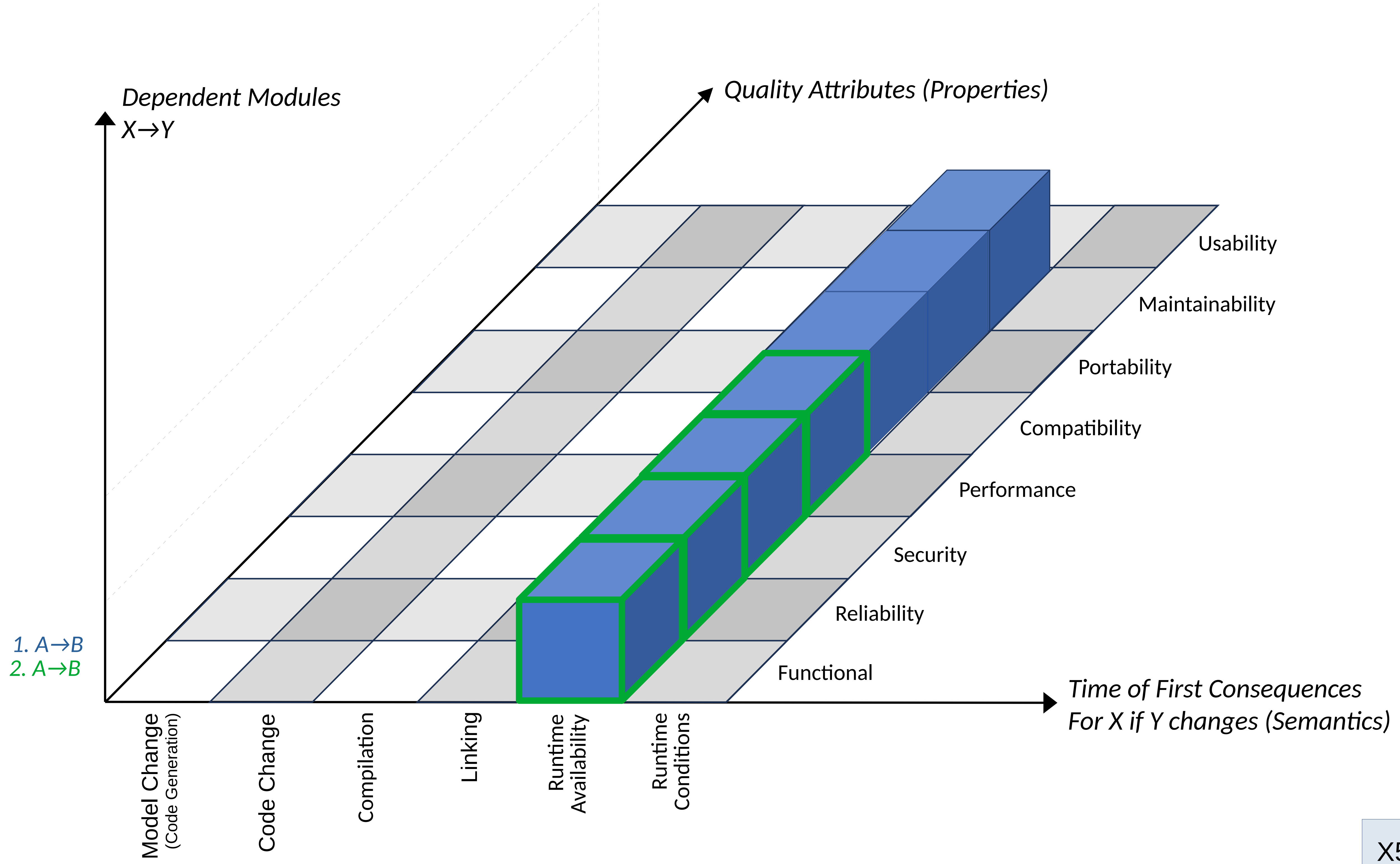
For example, if B is a C++ template library or defines inline functions.



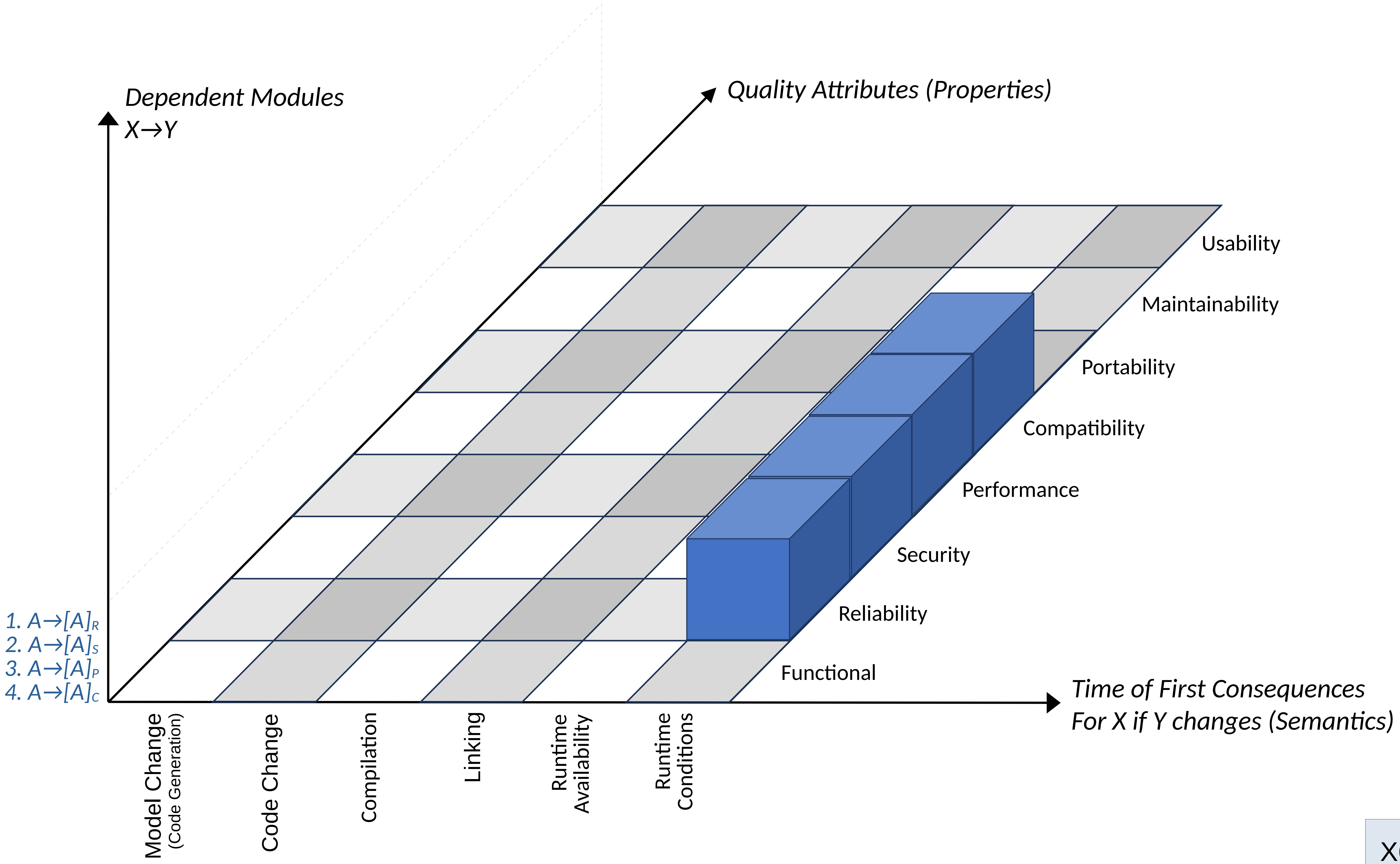
Module A depends on module B, which must be statically relinked when B changes.



1. Module A depends on module B, which is dynamically linked, so no relinking is required as long as the interface remains stable, but the new shared object (DLL) must be available on the machine.
2. Module A depends on module B, connected via IPC. No relinking is necessary, but the updated module B, which is expected to reside in a different executable/process, must be available at runtime.



Module A depends on different property groups, e.g.:
1. $[A]_R$: All modules running in same thread/process as A so that a fatal error leads to a crash.
2. $[A]_S$: All modules running under same user as A.
3. $[A]_P$: All modules sharing the same processing unit / cgroups as A.
4. $[A]_C$: All modules that can interfere with each other over a shared resource used by A (co-existence).



Along the y-Axis

Similar to the equivalence classes used in X6, we can also identify equivalence classes that, in order to fulfill a certain property, result in specific dependencies between modules across different phases of the *x-axis*. This type of group dependency among the members of such a class can rapidly become an actual dependency if one group member violates the property rules, much like the specific dependency of a road user on all other road users, which becomes an actual dependence when one driver ignores the right of way, leading to an accident.

Y1 To obtain **reliable** results for a system composed of a set of modules {A_i}, the source code must adhere to specific paradigms and rules, which also apply to those parts linked to the resulting binary. Servers and dynamic libraries on which module A depends must be available at runtime, and the runtime conditions must match the resource requirements.

Y2 To ensure a **secure** system composed of a set of modules {A_i}, the source code must adhere to specific rules, which also apply to those parts linked to the resulting binary. Servers and dynamic libraries on which module A depends must be available at runtime, connections must be secure, and the running context must be isolated from parts with a lower security level.

Y3 A **performant** system composed of a set of modules {A_i} needs efficient implementation, which also applies to all local libraries and remote contributors. The latter must be available at runtime via a performant connection, and the running context must ensure sufficient resources.

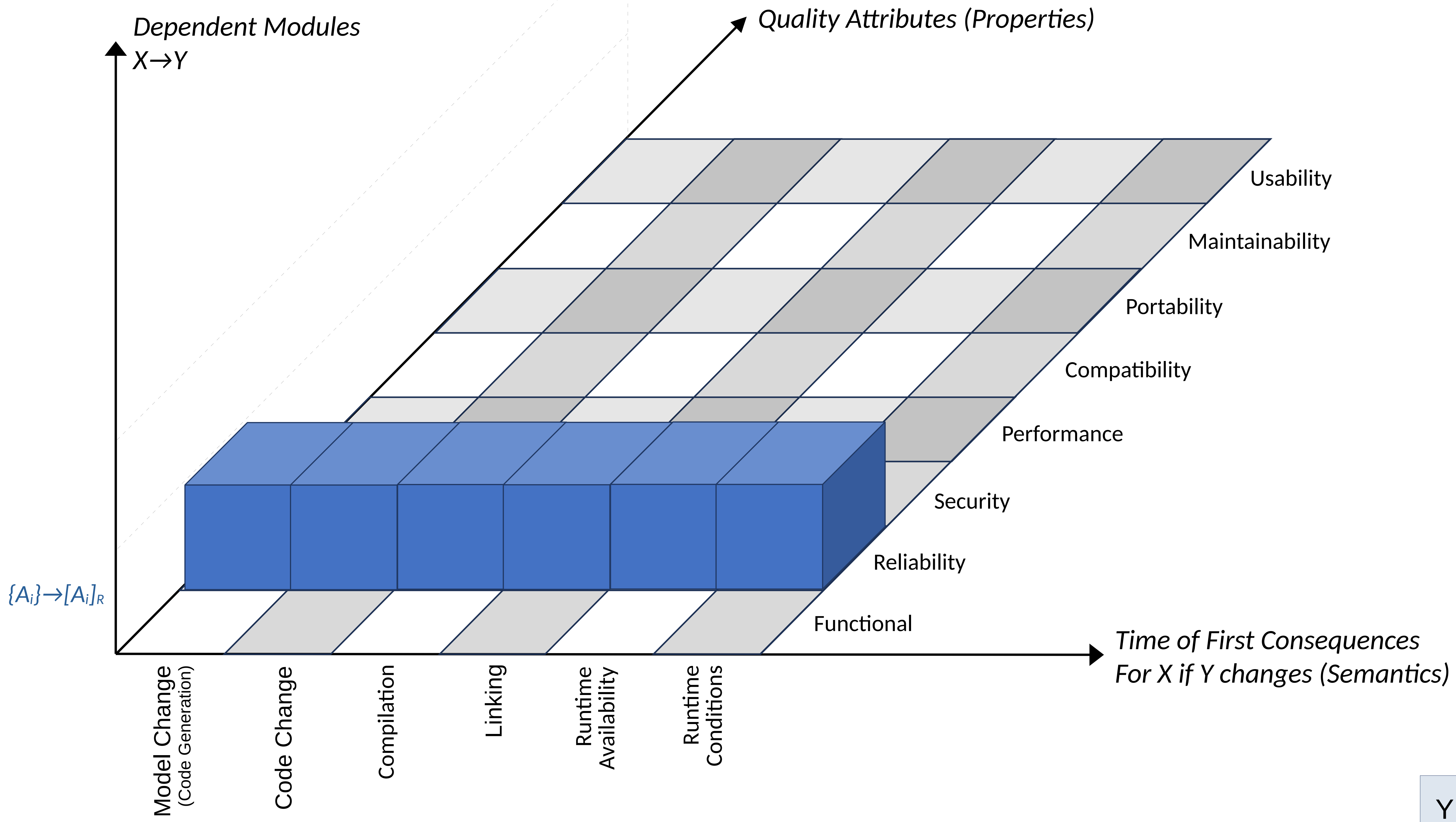
Y4 Like a performant system, a **compatible** system has to ensure the reasonable use of shared resources in all phases along the y-axis to make co-existence possible.

Y5 **Portability** considerations are already embedded in decisions related to the programming language and the choice of a compatible build toolchain. Additionally, dynamic libraries must be available in a compatible variant/version, and factors such as endianness must be addressed when communicating remotely. As soon as the system is up and running, portability is obviously guaranteed, runtime conditions have no influence here.

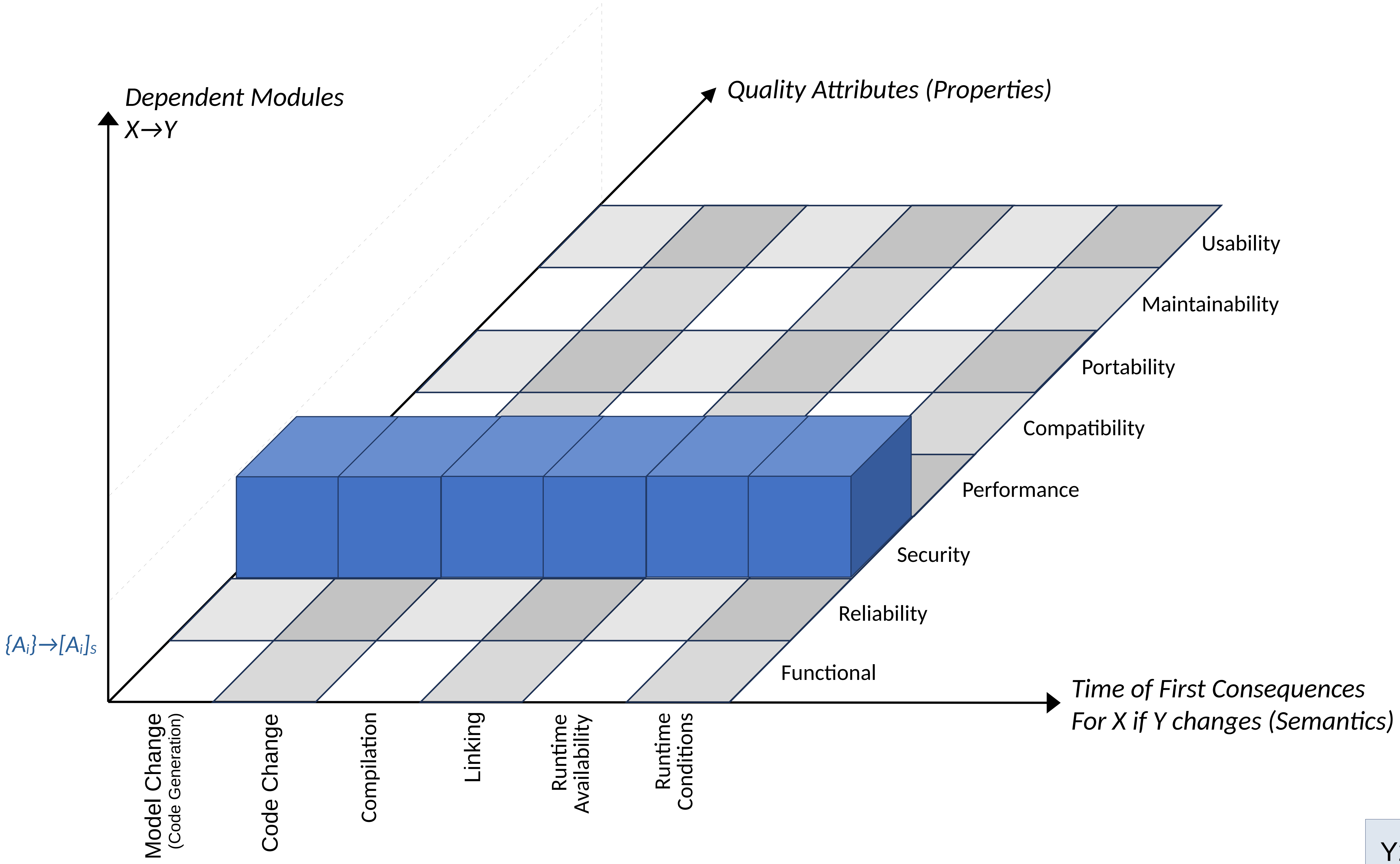
Y6 **Maintainability** goes hand in hand with modularity, i.e., how the system is divided into pieces or modules. This impact spans all phases, from model change to runtime availability of dynamic libraries or remote servers. In general, maintainability must be a part of the initial design; it is not something induced from external factors like runtime conditions.

Y7 Changing the API **usability** requires a code change and recompilation; only linking or runtime availability are not sufficient.

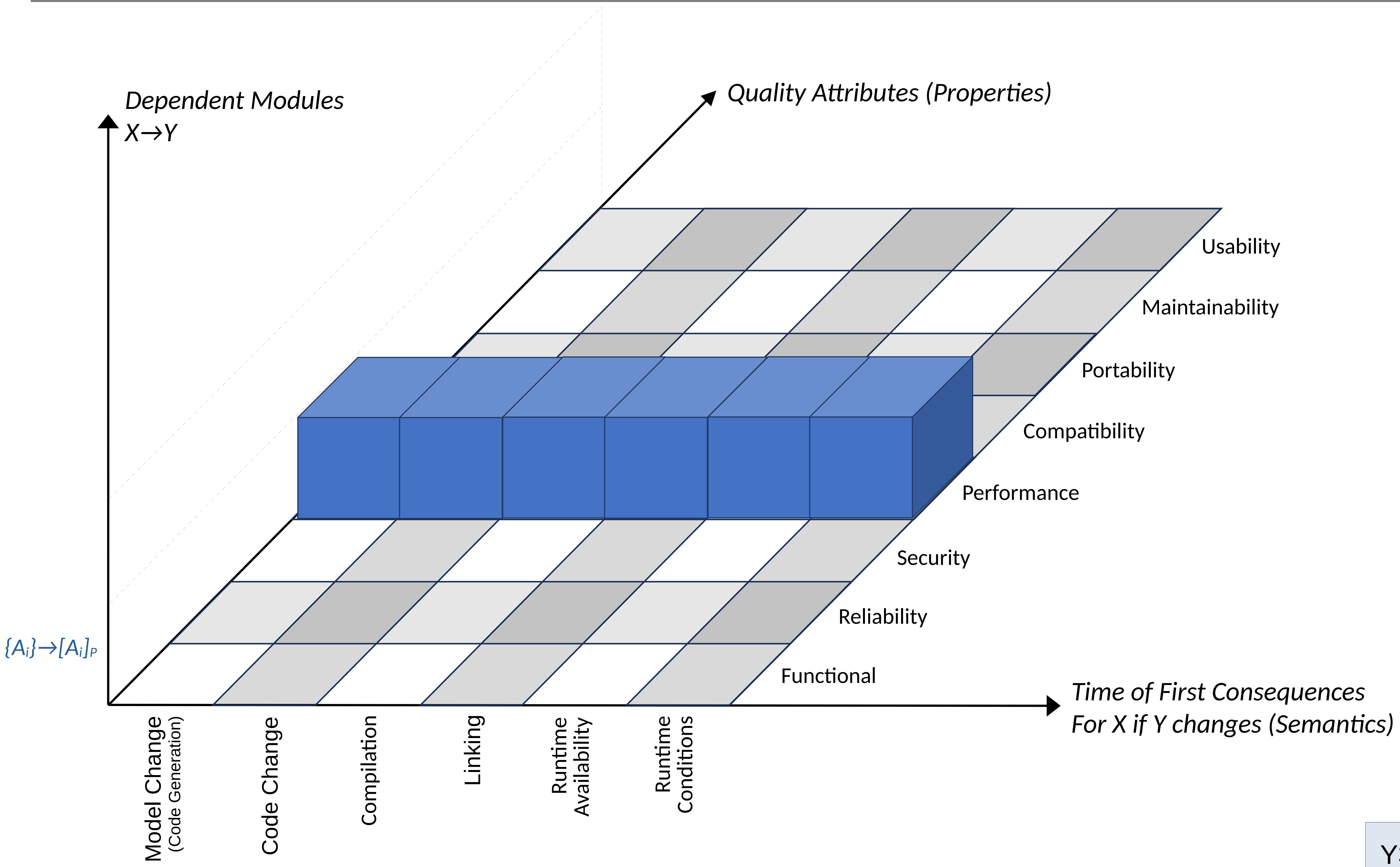
As system defined by all its modules $\{A_i\}$ depends in terms of reliability on the equivalence classes $[A_i]_R$, where the relation R describes all modules with mutual influence on reliability. For example, this includes all modules running in the same thread/process as A_i, where a fatal error in any of them leads to a crash.



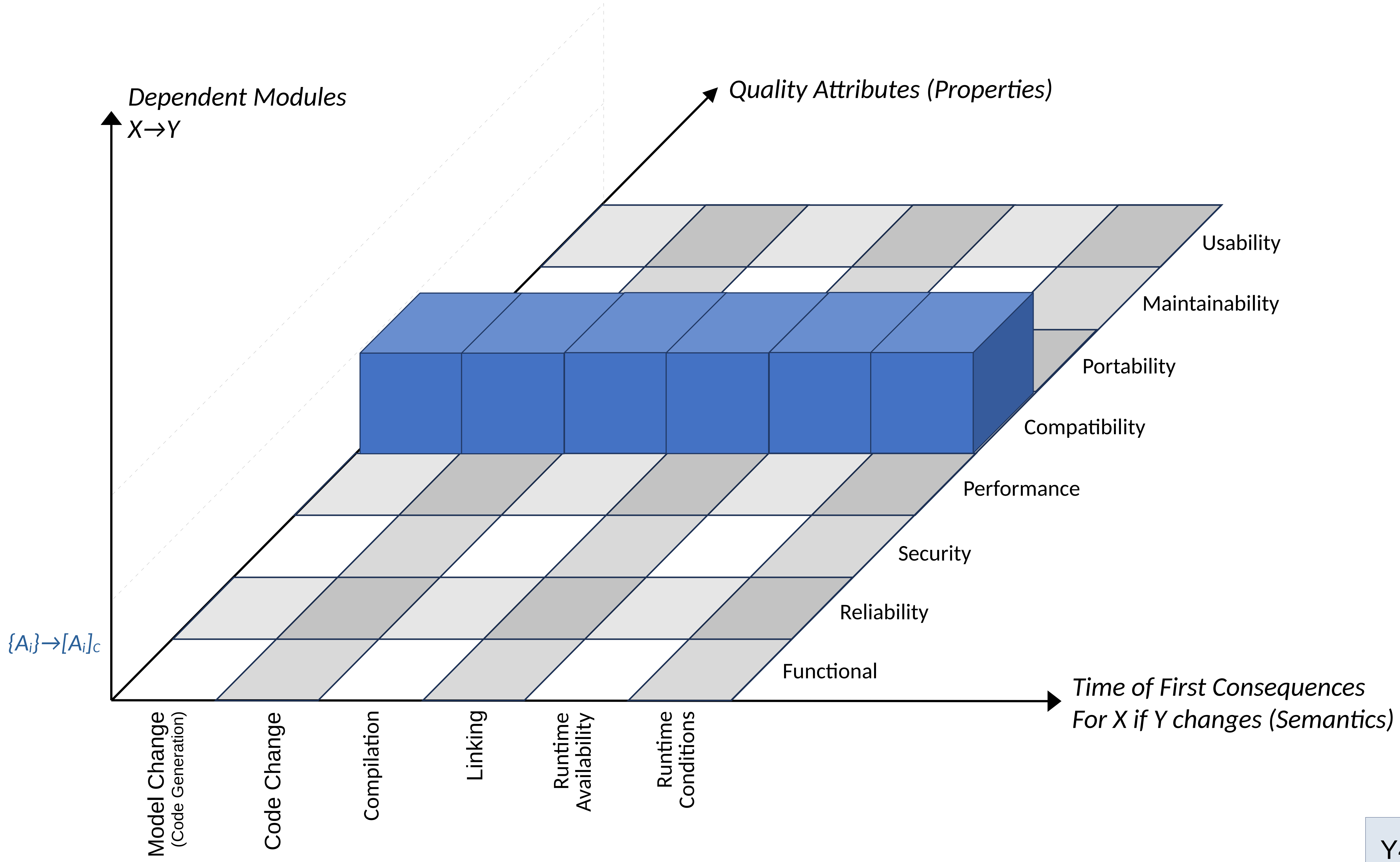
As system defined by all its modules $\{A_i\}$ depends in terms of security on the equivalence classes $[A_i]_S$, where the relation S describes all modules with mutual influence on security. For example, this includes all modules running under same user as A_i.



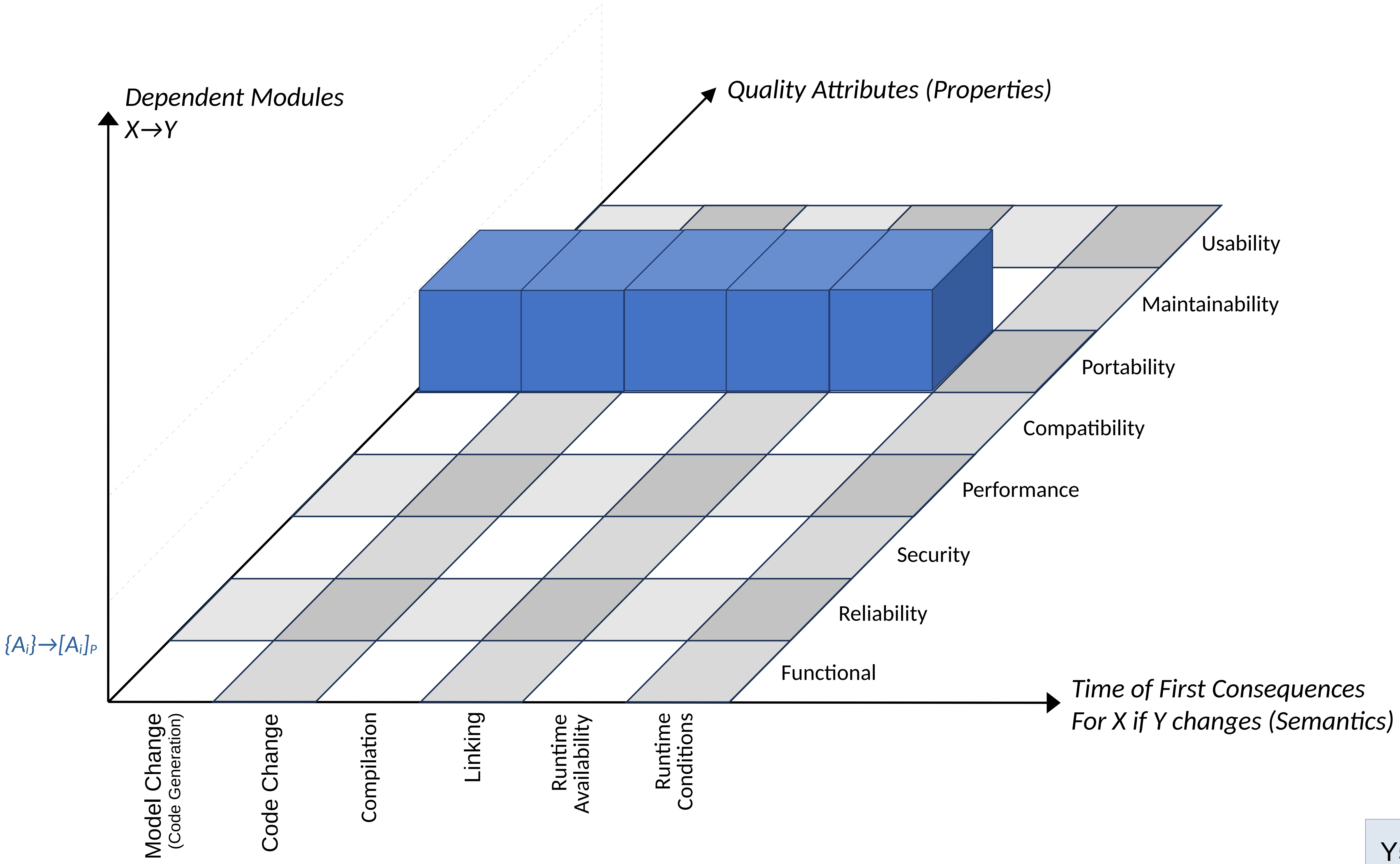
As system defined by all its modules $\{A_i\}$ depends in terms of performance on the equivalence classes $[A_i]_P$, where the relation P describes all modules with mutual influence on performance. For example, this includes all modules sharing the same processing unit / cgroups as A_i.



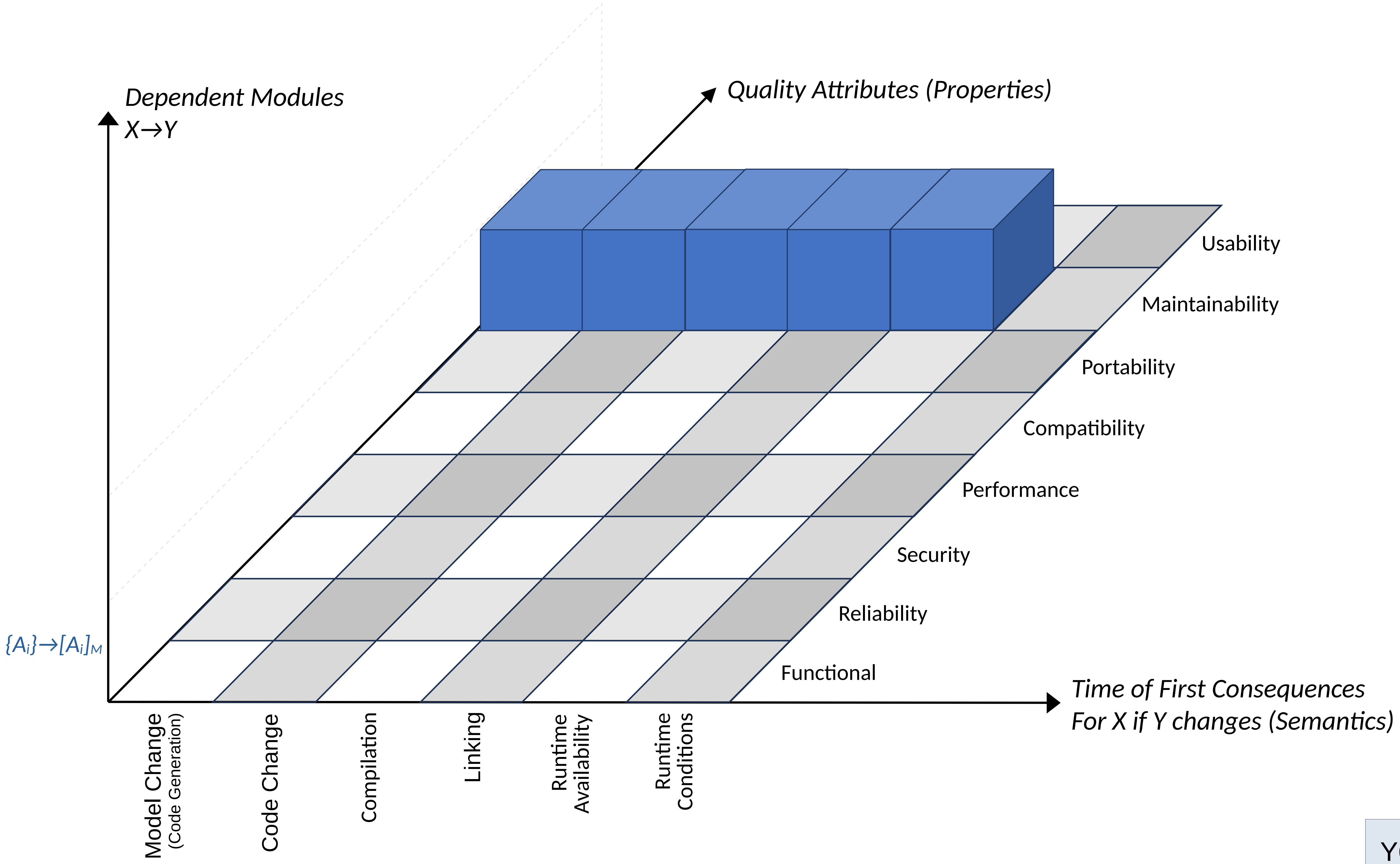
As system defined by all its modules $\{A_i\}$ depends in terms of compatibility on the equivalence classes $[A_i]_C$, where the relation C describes all modules with mutual influence on compatibility. For example, this includes all modules that can impair each other over a shared resource used by A_i (co-existence).



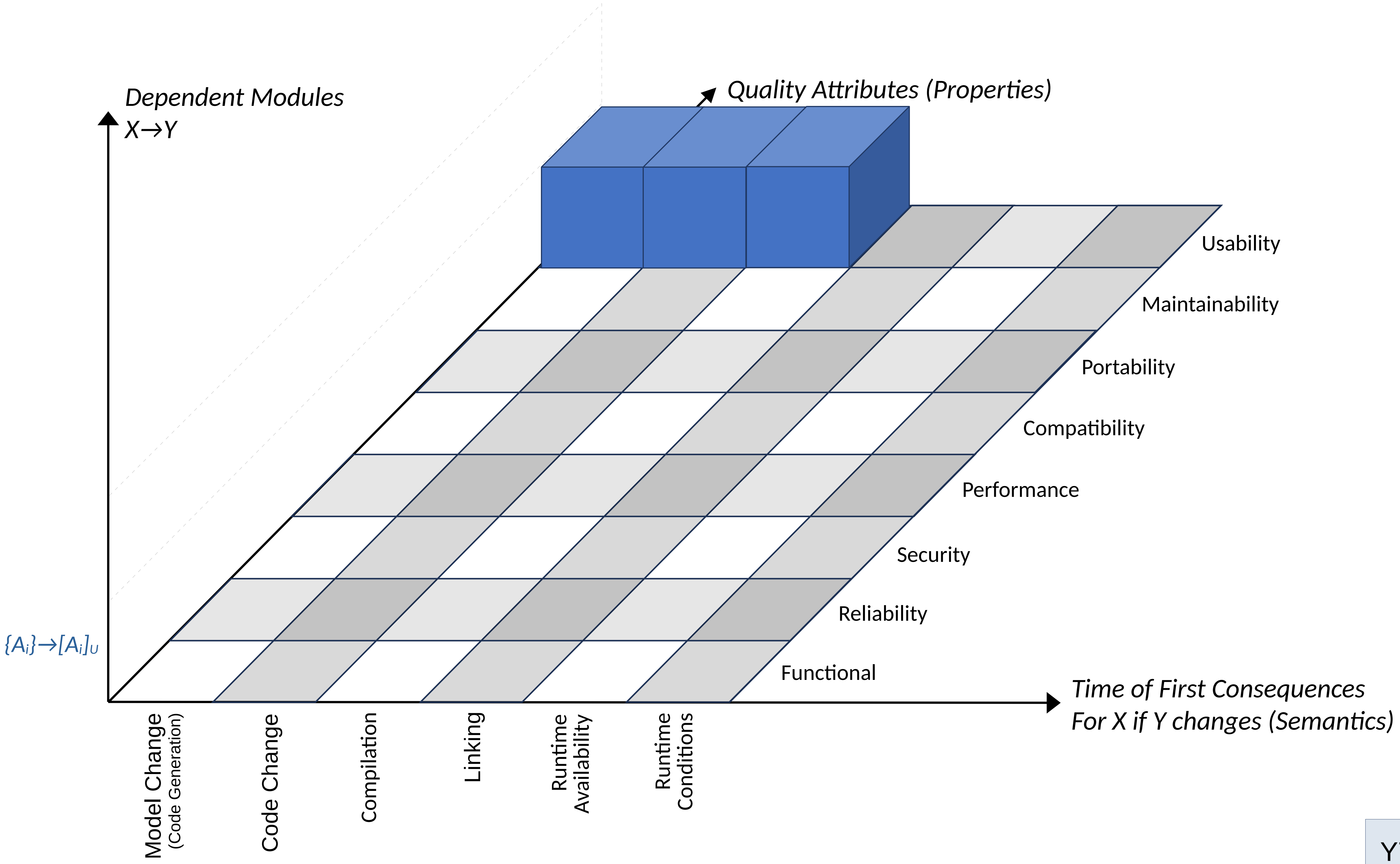
A system defined by all its modules {A_i} depends in terms of portability on the equivalence classes $[A_i]_P$, where the relation P describes all modules with mutual influence on portability. For example, this includes all modules programmed with the same programming language and depending on the same or at least compatible build toolchain (which also applies to dynamic libraries). In the case of server connections, both sides have to agree on endianness, for instance.



A system defined by all its modules $\{A_i\}$ depends in terms of maintainability on the equivalence classes $[A_i]_M$, where the relation M describes all modules with mutual influence on maintainability. For example, this includes all modules that are part of one delivery package and, therefore, need to be aligned in schedule and declaration of external dependencies.



A system defined by all its modules $\{A_i\}$ depends in terms of portability on the equivalence classes $[A_i]_U$, where the relation U describes all modules with mutual influence on API usability. For example, this includes all modules that provide a portion of a public API to the customer and, therefore, should adhere to a common look, feel and style.



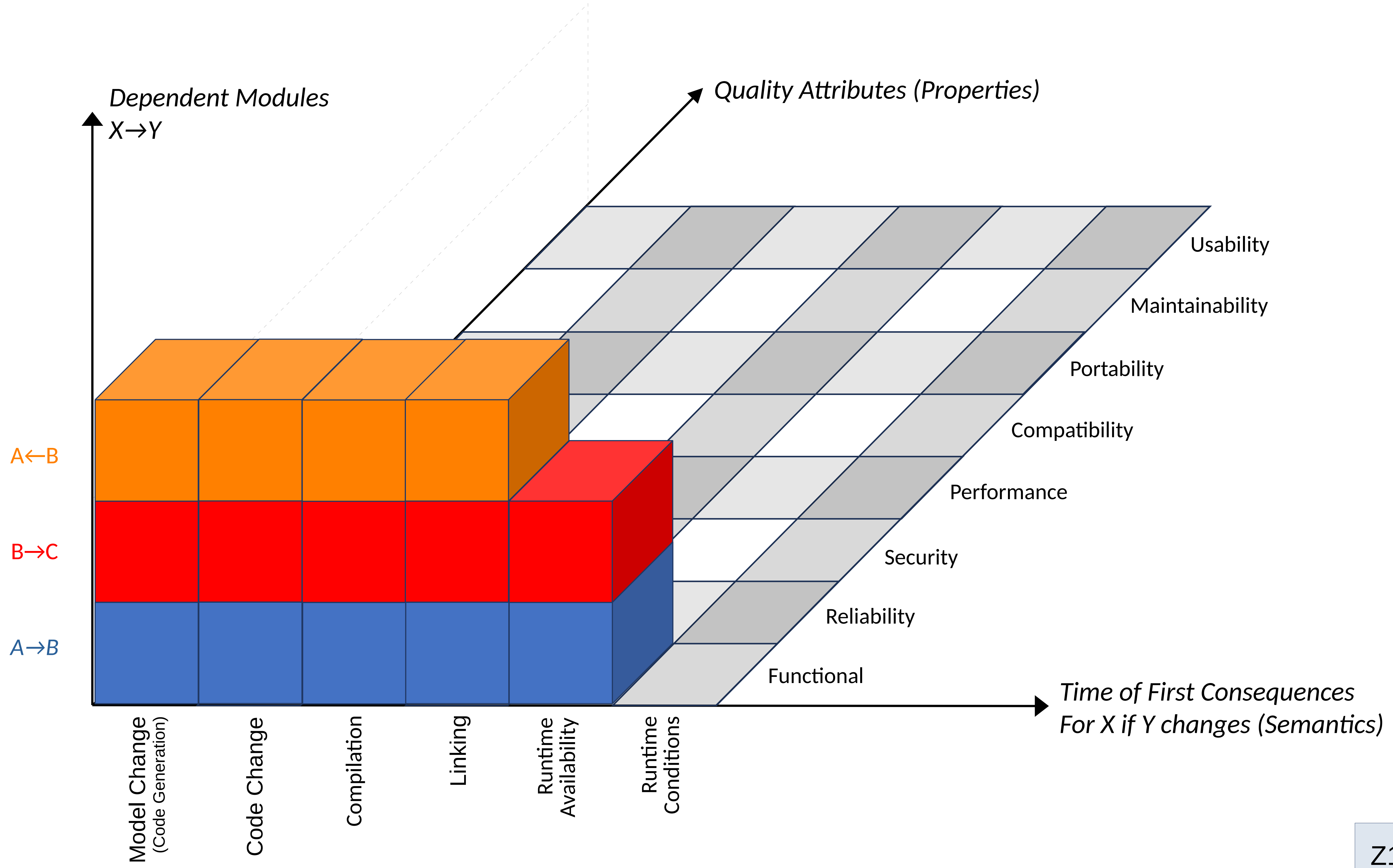
Dependent Modules $X \rightarrow Y$
Quality Attributes (Properties)
Usability
Maintainability
Portability
Compatibility
Performance
Security
Reliability
Functional
$\{A_i\} \rightarrow [A_i]_U$
Time of First Consequences For X if Y changes (Semantics)
Model Change (Code Generation)
Code Change
Compilation
Linking
Runtime Availability
Runtime Conditions

The z-Axis

Z1 Along the z-axis, the dependencies between different modules are listed. Since the 2D projection used in this article can quickly become overloaded and confusing when compared to a genuine 3D illustration, it does not make sense to display more than three layers in a single diagram. While this may be insufficient to depict all dependencies of a system, it is adequate when focusing on a specific aspect of one module.

Special attention should be given to **Permitted Changes** to ensure that a change does not introduce new unwanted dependencies or undesirable **Cyclic Dependencies**. Monitoring can be based on the foremost vertical pane that represents *Functional Suitability*, as dependencies are initially introduced due to functional needs.

If you change module B, on which module A depends (A→B) …
… be careful that no new (unwanted) dependency to module C is added (B→C).
… void introducing a cyclic build-time dependency from B to A (A←B). However, A can play ping-pong with a remote server D if you like.



The Puzzles

The puzzles like defined by Jimmy Koppel in his article “*How an Ancient Philosophy Problem Explains Software Dependence”, for details refer to* [Koppel22].

P1 **The Inversion Puzzle**: Does Dependency Inversion truly eliminate dependencies? Well, there is a reason why it is called inversion, see P1a and P1b.

P2 **The Framework Puzzle**: Does a Round-Robin scheduler have dependencies on the tasks it invokes? At least that task has to exist, see P2.

P3 **The Shared Format Puzzle**: If W writes a file and R reads it, both are coupled due to the shared format. But which module depends on the other? Do not tear apart what belongs together, see P3.

P4 **The Fallback Puzzle**: When A uses B but falls back to C in the event of B's failure, does A still depend on B? Dependencies cannot only exist on specific modules but also on groups (sets and equivalence classes) of modules, see P4.

P5 **The Majority Puzzle**: How does dependency work in the case of m-out-of-n redundancy? Same answer as for The Fallback Puzzle, see P4.

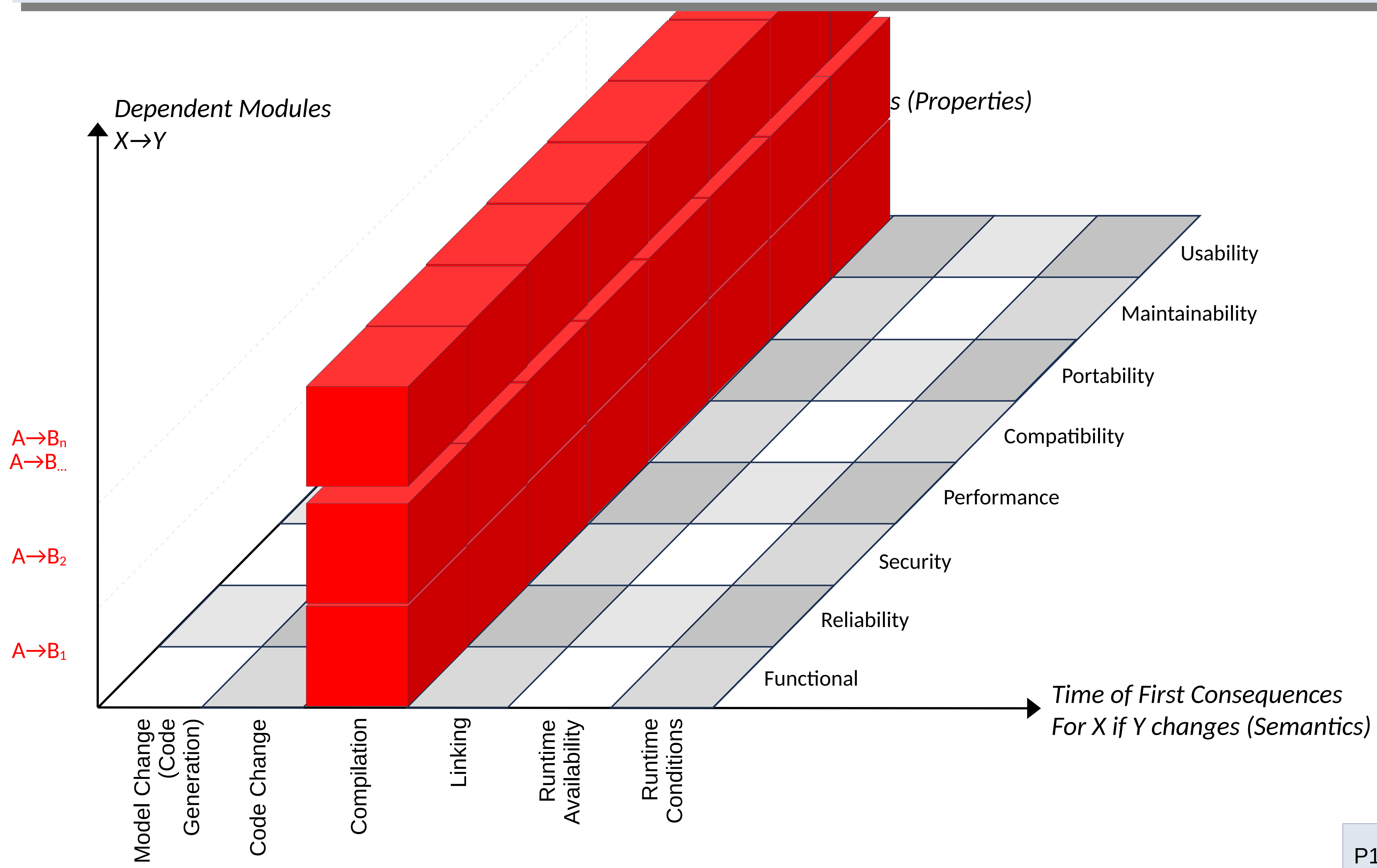
P6 **The Higher-Order Puzzle**: Do hash tables and other library classes depend on their callers? Dependency Inversion again, refer see P1b.

P7 **The Crash Puzzle**: In principle, each module of a program can crash it. Does this imply that each module depends on every other module? Already discussed in X6.1.

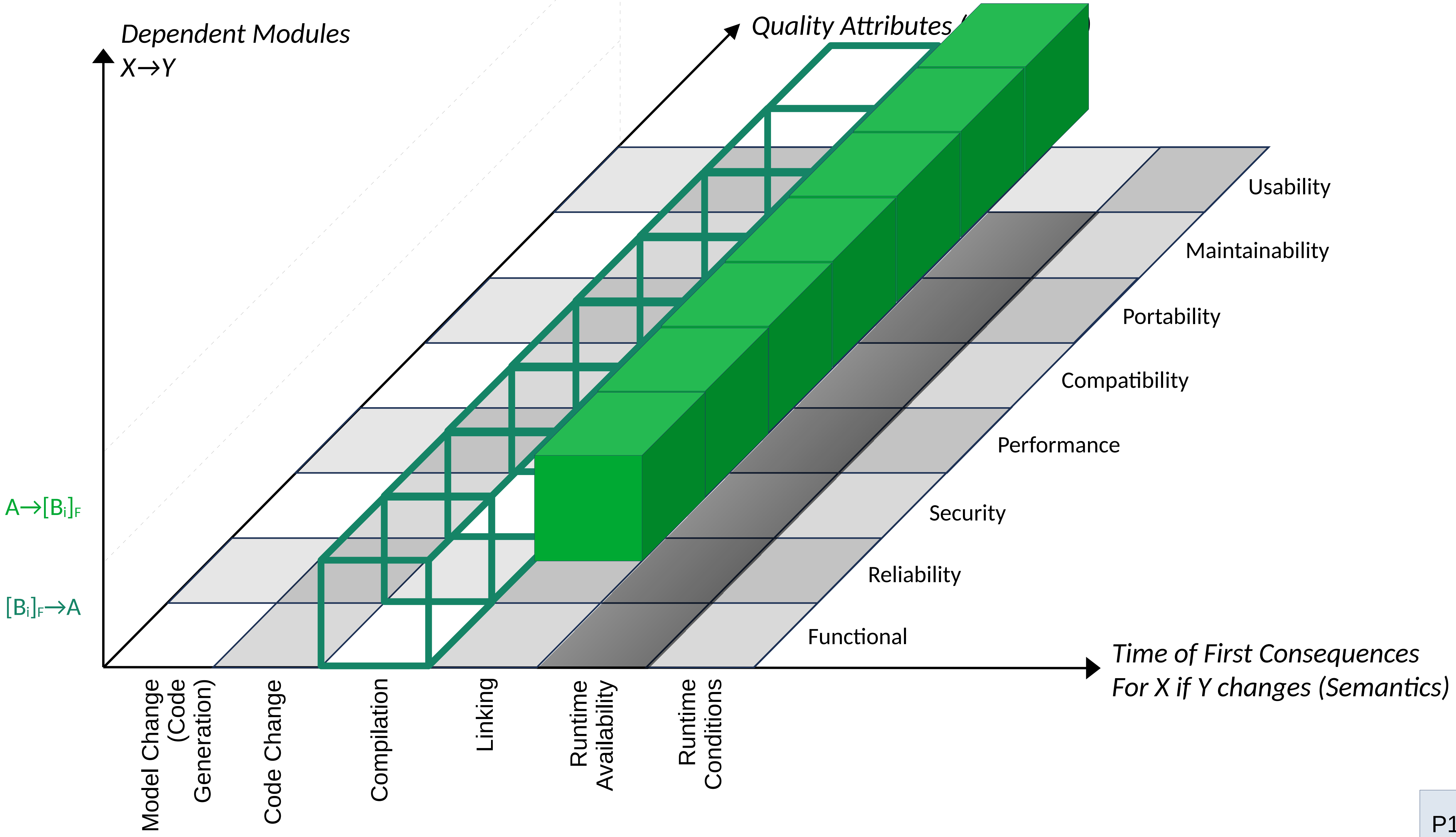
P8 **The Self-Cycle Puzzle**: How to deal with cyclic dependencies and self-dependencies? Refer to Z0.

P9 **The Frame Puzzle**: If A does not depend on B, does a change to B affect A? What if that change introduces a dependency on A? Refer to Z0.

Dependency Inversion I:
Module A depends on the implementation of several modules B_i. If the functionality F behind these dependencies can be formulated as an abstract interface defined by A, which must be implemented by all $\{B_i\}$ ...



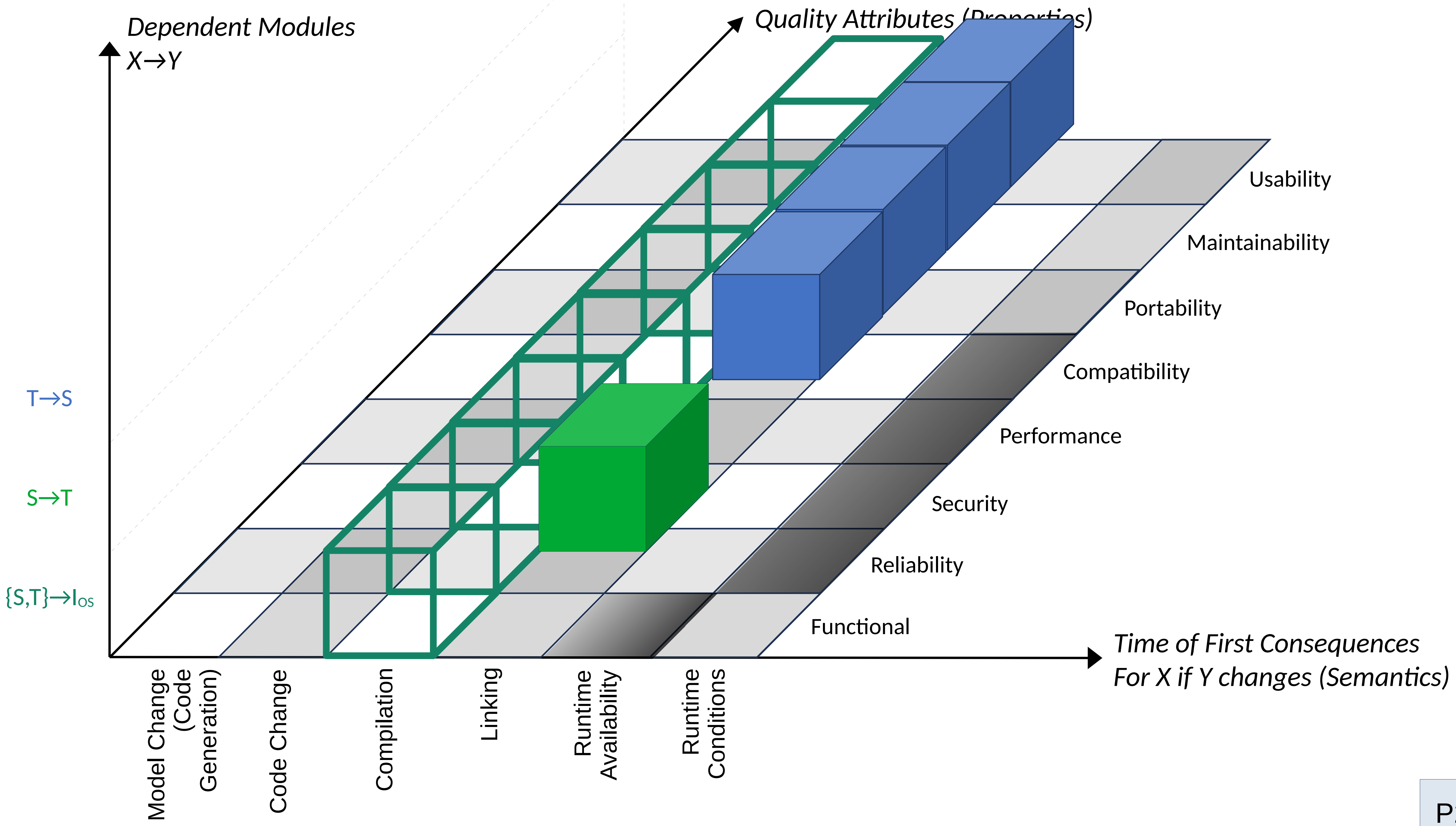
Dependency Inversion II:
... then this multiple dependency can be converted (inverted) into $[B_i]_F \rightarrow A$. However, at runtime, A still depends on the implementation $A \rightarrow [B_i]_F$.



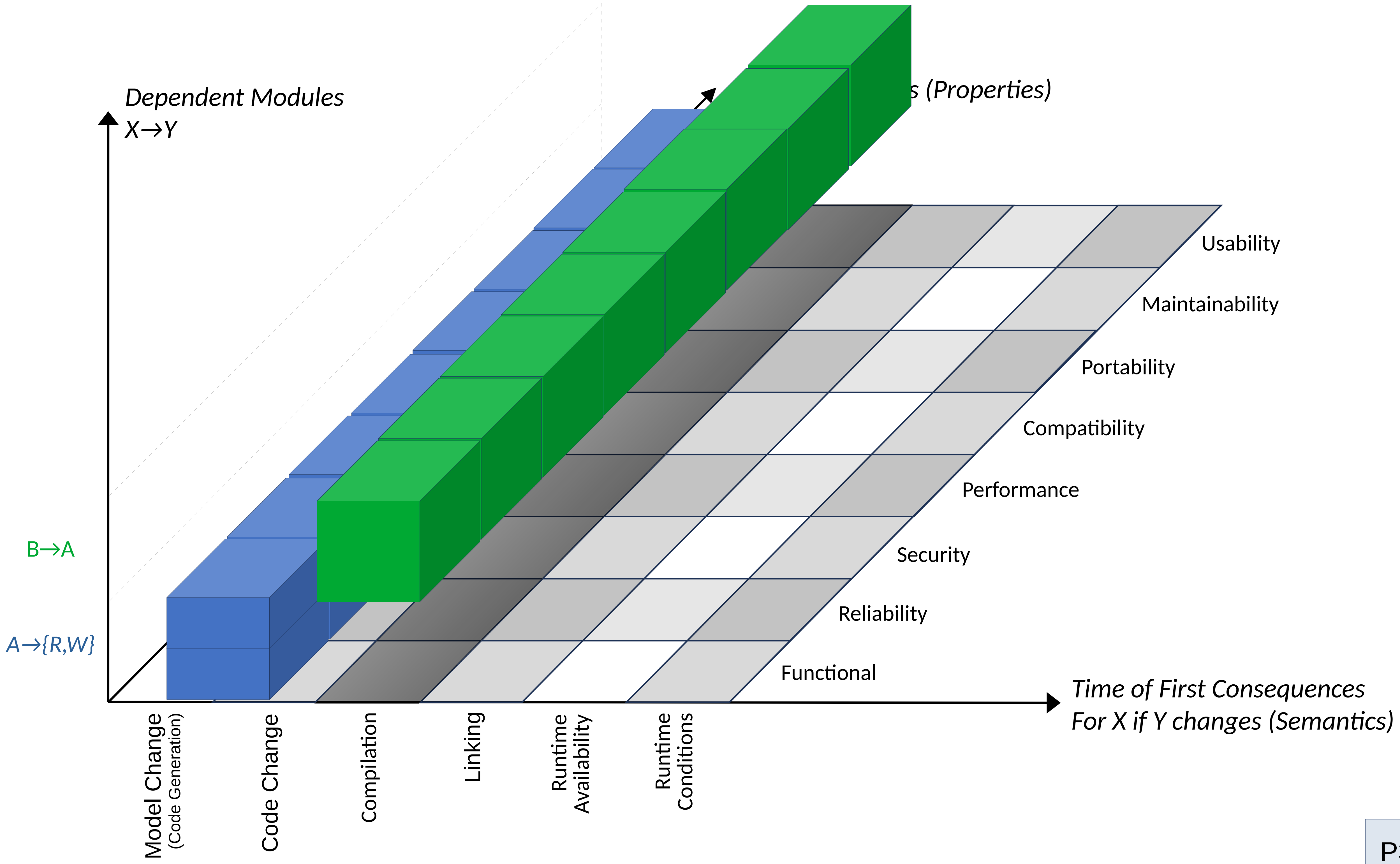
P1b

Round-Robin Scheduler:

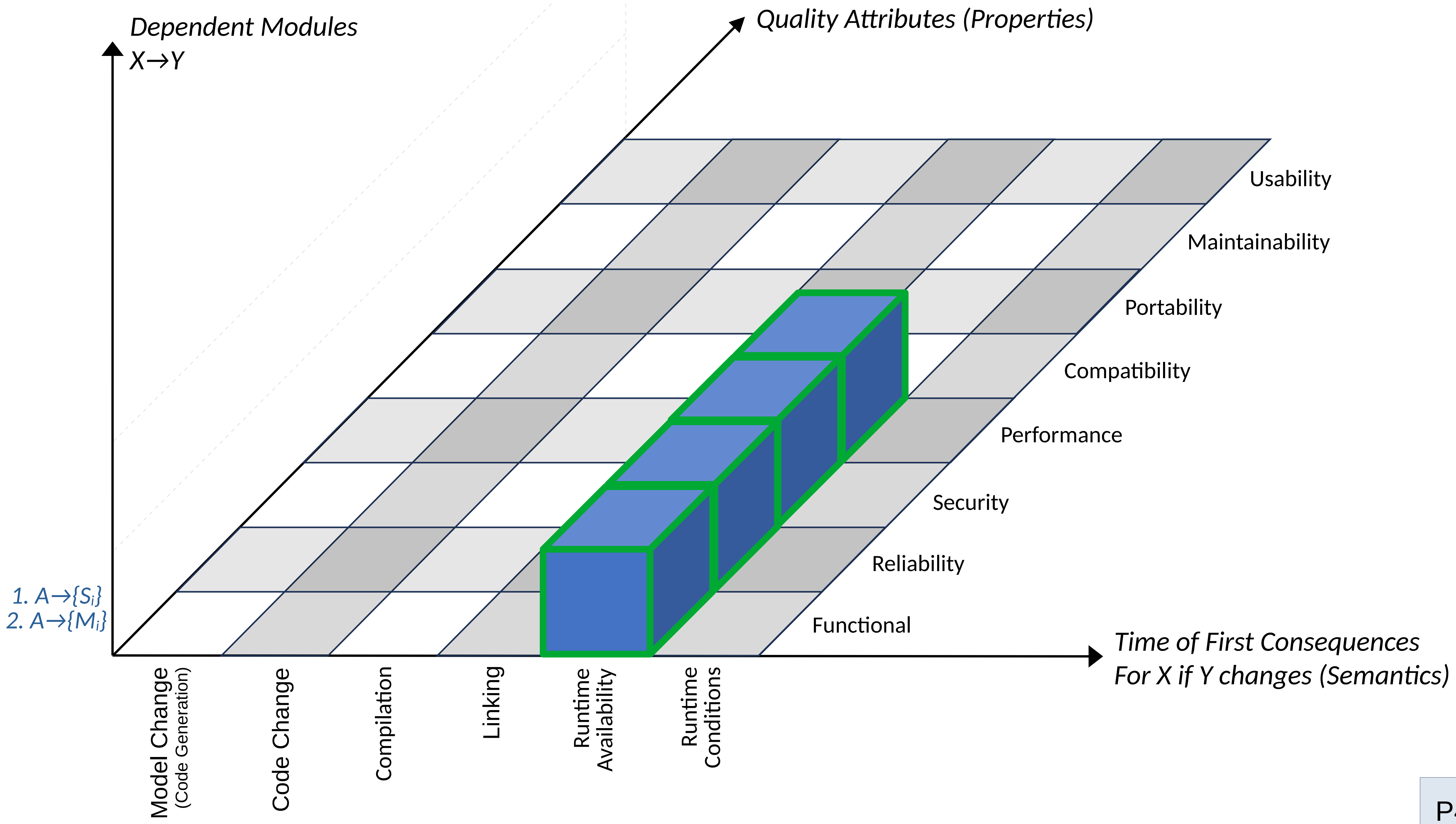
Special case of Dependency Inversion (P1b): Both the scheduler S and the task T depend on the task management interface I_{OS} of the OS. The scheduler S is shielded by the OS to protect against potential reliability, security, performance, and compatibility impacts introduced by T. However, T's runtime conditions depend on the scheduler S.



Reader/Writer (Sub-)Modules:
The reader/writer (serializer/deserializer) exists as an abstract model that can be used to generate source code (see X1) or as part of a source code module (see X2). In either case, R and W should be part of one module A that can be used by any module B.



1. Fallback:
   Module A depends on one out of n redundant sources S, as at least one source out of n must be available, $\{F_i\} \subseteq \{S_i\}$, $|\{F_i\}| > 0$.
2. Majority:
   Module A depends on one more than half of the n sources S, as the majority of the n sources must yield the same result, $\{M_i\} \subseteq \{S_i\}$, $|\{M_i\}| > 0.5\ |\{S_i\}|$.



P4

References

[Wolff23]	Eberhard Wolff, *Software Architektur im Stream: Folge 179 - Software-Architektur = Abhängigkeiten Managen?* https://software-architektur.tv/2023/09/01/folge179.html
[Koppel22]	Jimmy Koppel, *How an Ancient Philosophy Problem Explains Software Dependence*, 2022/09/30, https://www.pathsensitive.com/2022/09/bet-you-cant-solve-these-9-dependency.html
[ISO]	ISO/IEC 25010, https://iso25000.com/index.php/en/iso-25000-standards/iso-25010