Key Modeling Principles to Moderate the Growth of Model Technical Debt in MBSE

Ryan Noguchi
The Aerospace Corporation

2024 NDIA Systems & Mission Engineering Conference
October 2024

Approved for public release. OTR-2024-01193.

The Technical Debt Concept

- Technical debt is a well-known concept in the software domain:
  - A metaphor for development or sustainment costs that have been deferred to the future (Cunningham, 1992)

- Technical debt represents the deferred cost of repair, rework, or replacement of a product that wasn't built perfectly from the beginning
  - Technical debt is "created when developers violate good architectural or coding practices, creating structural flaws in the code" (Curtis et al. 2012a)

- Much research and evolution of practice in software development is focused on managing technical debt
  - Reducing its growth
  - Burning it down

Cunningham, W. (1992). The Wycash portfolio management system, in Addendum to the Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM Press.

Curtis, B., Sappidi, J., & Szynkarski, A. (2012a). Estimating the Size, Cost, and Types of Technical Debt. 3rd International Workshop on Managing Technical Debt.

The technical debt concept from software is valuable for descriptive models… with modification

Principal in Technical Debt

- In financial debt, **principal** is the amount of capital borrowed
  - i.e., the cost to purchase something now to avoid incurring debt

- In software, the equivalent of principal is the additional cost of creating an optimal implementation over creating a suboptimal implementation
  - i.e., principal represents the cost to “do it right,” paying the full cost now rather than deferring that cost

***Principal* is the cost of “doing it right” now, which one is deferring for whatever reason**

Interest in Technical Debt

- The technical debt equivalent of **interest** is the additional cost needed to rework the implementation to accommodate the current context
  - i.e., it represents the cost to “make it right” by paying off the debt later
- The interest that needs to be paid typically **increases with time**
  - As the implementation grows, the amount of rework needed also grows due to replication and propagation of dependencies
- Rework cost can manifest in greater difficulty to **extend**, **maintain**, or **evolve** the implementation
- However, it is also possible that the debt never needs to be repaid
  - i.e., the rework is not needed, since that component is no longer needed
  - Or the context has changed sufficiently to avoid the need for rework

***Interest* is (part of) the cost of deferring full payment until later**

Taxes in Technical Debt

- In the financial domain, **taxes** often manifest as added costs on transactions

- In the software domain, taxes represent additional costs imposed on the day-to-day work of developers or users that result directly or indirectly from the suboptimal implementation
  - Having to “work around” convoluted code
  - Users suffering clunky or unintuitive user interfaces
  - Unlike *interest*, these are generally **transactional** costs
    - Imposed only when developers or users need to use that code
  - Result in **lost productivity** and **lost value**

- Often, in the software domain, the notions of interest and taxes are conflated or combined
  - However, these possess very different mechanisms and patterns of growth and manifest very differently to developers and users

***Taxes* are transactional costs resulting from suboptimal implementation**

Why—or Why Not—Technical Debt?

- The goal is **not really to eliminate debt**

- Rather, the goal is to make better-informed choices about what debt to take on or to pay off
  - Debt is often a good investment, providing opportunities that would otherwise be unavailable
  - Accrued debt can be paid out sooner or deferred

- Unlike financial debt instruments, technical debt is generally not deterministic
  - What constitutes technical debt will **change over time** as the **context changes**
  - If a portion of the product is eliminated, the rework associated with that portion may never materialize
  - Was once technical debt may no longer be as significant
  - What wasn't technical debt before has now become technical debt

- Therefore, technical debt is a fluid characterization that evolves over time because of the continually evolving context and its implications on the need for payment of the debt

Today's technical debt might appear or disappear overnight

The (Different) Calculus of Technical Debt in Descriptive Models

- The cost of rework of models is often substantially greater than it is for software
  - Model rework is more often architectural in scope and nature
  - Model rework is more pervasive, invasive, interdependent, and difficult to perform than with software
  - Implementation details of descriptive models are fully exposed to users
  - Refactoring of models is much less amenable to piecemeal scheduling
    - The model is internally inconsistent from the start of the first change to the end of the last change
  - Automated discovery of need for refactoring is often difficult
    - Particularly when driven more by semantics than syntax

- If a model defect is not corrected, it can multiply and spread to other parts of the model
  - In the software technical debt domain, this is called “contagion” (Bi at al., 2021; Martini & Bosch, 2015)

Bi, F., Vogel-Heuser, B., Huang, Z., & Ocker, F. (2023). Characteristics, Causes, and Consequences of Technical Debt in the Automation Domain. Journal of Systems and Software, v. 204.

Martini, A. & Bosch, J. (2015). The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles. 12th Working IEEE/IFIP Conference on Software Architecture.

The increased risk and impact of technical debt in descriptive models results in greater value of cautious architectural decision making to control the growth of technical debt more conscientiously

Characterizing Model Technical Debt

- The growth of interest and taxes as the model matures can be characterized by a few critical factors:

Scale/Growth	• How many model elements are affected? • Are these very common items? • Are these used in many different ways in the model?
Propagation	• How quickly will these impacts indirectly propagate through the model? • How quickly do the impacts “jump species” in the model?
Rework	• How difficult is executing the rework? • How difficult is finding what needs to be reworked? • How independent is the rework process?
Use Taxes	• What are the impacts or taxes? • Is the impact isolated in time or recurring many times?

These factors influence the risk assumed when one makes architectural or implementation decisions that reflect the potential of increasing technical debt

Modeling Principles to Mitigate the Growth of Technical Debt

- The remainder of this presentation identifies the key principles that model architects should consider when making architectural and implementation decisions about their models

- It also describes key model technical debt tradeoffs that result when these principles are not followed

- Each principle is provided a subjective rating—ranging from Low to High—representing the commonly observed impacts of failing to follow that principle

- These impacts are characterized by
  1. the **growth** of interest as the model grows
  2. the **propagation** of interest through the model
  3. the difficulty of **rework**, manifesting as increased interest when the debt must be repaid
  4. the recurring impacts (**taxes**) on model users until the problematic aspect has been corrected

The goal is to enable model architects to make better decisions about technical debt tradeoffs for their models

1. Model for Purpose

- One of the biggest obstacles to modeling success is the lack of well-understood, explicitly defined and documented, sufficiently compelling purposes for the model to fulfill

Model Implementation Principle	Technical Debt Implications
1. Model for Purpose: Establish sufficiently valuable purposes for the models as early as possible	Redundant or duplicative information in the model is difficult to cleanly remove and can lead to incorrect or misleading answers to model queries. Scale: High | Propagation: High | Rework: High | Use Tax: High

- Without a good understanding of a model's intended purposes, the model architect can't make well-informed model architectural decisions

- Not properly addressing model users' information needs frequently results in substantial model churn
  - The model needs to be continually reworked to become useful
  - The model can become too costly to be worth refactoring—may be less work to start over

- Hence, understanding the most essential functional needs for the model is vitally important
  - Drives the early model architectural decisions that can significantly influence its chances of long-term success

Model Federation Principles and their Technical Debt Implications

- A model federation is a distributed set of models connected by a controlled set of model usage relationships
  - To enable their content to be shared whilst retaining their autonomy and ability to evolve independently
- Federation adds some complexity and requires active and conscientious architecting
  - To successfully achieve the best tradeoffs among competing alternative federation architectures and approaches

Model Federation Principle	Technical Debt Implications
2. Consider Federation Early: Consider model federation architecture as early as possible.	Model federation architecture becomes increasingly difficult to change as the model grows in size and use. Scale: High | Propagation: High | Rework: High | Use Tax: Low
3. Partition for Cohesion: Partition models to align with content and usage boundaries.	Misalignment with content boundaries complicates model governance and sustainment; misalignment with usage boundaries complicates model use. Scale: High | Propagation: High | Rework: High | Use Tax: High
4. Federate to Interrogate: Establish directional model usages to align with user queries.	Query navigation paths that are in opposition to model usage paths makes it difficult to use the model to answer questions. Scale: High | Propagation: High | Rework: High | Use Tax: High
5. Dependency Inversion Principle: Direct dependencies toward increased abstractions.	More concrete models typically experience substantially greater frequency and depth of change, which impacts models that depend on those models. Scale: High | Propagation: High | Rework: High | Use Tax: High
6. Define Clear Interfaces: Architect well-defined interfaces between federated models.	Failing to clearly define model interfaces often results in inconsistent inter-model connections and encapsulation breaking. Scale: High | Propagation: High | Rework: High | Use Tax: High

Model Layers Principles and their Technical Debt Implications

- Descriptive models are constructed with multiple layers
  - **Layers of abstraction** facilitate separations of concerns
  - **Taxonomic layers** represent the key concepts of the domain at different levels of granularity
- The model architect needs to make critical decisions regarding the definition of these layers
  - Selecting the right number of layers
  - Identifying the intent of each layer
  - Defining the interfaces between layers

Model Layers Principle	Technical Debt Implications
7. Just Enough Layers: Create just enough layers of abstraction or taxonomy—no more, no less.	Too few layers produce technical debt that must be repaid when layers must be added later. Too many layers complicate model growth and maintenance. **Scale: Medium** | **Propagation: High** | **Rework: High** | **Use Tax: Low**
8. Don't Cross Streams: Avoid mixing abstraction or taxonomic layers.	Mixing abstraction and taxonomic levels in the same hierarchy, or uncontrolled connections between levels, results in difficult use and painful rework. **Scale: Medium** | **Propagation: High** | **Rework: High** | **Use Tax: Low**
9. Scale-Free: Prefer scale-free decomposition over defining a specific hierarchy of levels.	Defining a specific hierarchy of levels often adds unnecessary complexity (additional layers) and can be overly constraining. **Scale: Medium** | **Propagation: High** | **Rework: Medium** | **Use Tax: Medium**

Modeling Domain Principles and their Technical Debt Implications

- Descriptive models for MBSE need to accurately represent the domain being modeled, at least to the extent that it answers the questions stakeholders want the models to answer

Modeling Domain Principle	Technical Debt Implications
10. Other People's Profiles: Leverage metamodel/profile standards in lieu of proliferating unique profiles.	Reinventing the wheel results in the need for continued investment in maintaining the unique profile over time and reduces interoperability. Scale: High | Propagation: High | Rework: High | Use Tax: High
11. Single Responsibility Principle: Type definitions should be minimal in scope.	Refactoring to break up combination concepts into more primitive concepts can be time consuming and error prone. Scale: Medium | Propagation: Medium | Rework: Medium | Use Tax: Low
12. Open/Closed Principle: Type definitions should be designed for specialization without modification.	Overconstrained base classes need to be reworked to be applicable to new contexts. Scale: Medium | Propagation: Medium | Rework: Medium | Use Tax: Low
13. Liskov Substitution Principle: Every specialization of a classifier must be a valid substitute for that classifier.	Violating LSP often results in confusion in the intent and proper use of both generalizations and specializations and short-circuits benefits of OO. Scale: Medium | Propagation: Medium | Rework: Medium | Use Tax: Medium
14. Interface Segregation Principle: Type definitions should not define features not used by specializations.	Violating ISP often introduces violations of other principles (like LSP) as workarounds and deactivating features can be difficult to implement. Scale: Medium | Propagation: Medium | Rework: Medium | Use Tax: Low
15. Reify to Reuse. Favor reification over other modeling constructs when concepts need to be reused.	Avoiding reification often results in accumulation of more primitive types, which are more prone to typographic errors and impede model queries. Scale: Medium | Propagation: High | Rework: High | Use Tax: Medium

Modeling Semantics Principles and their Technical Debt Implications

- Some of the most common but least visible problems in models are semantic in nature
  - Inconsistent use of modeling constructs, resulting in ambiguity
  - Mismatches between the modeler's and model users' expectations of the semantics of the concepts being modeled
    - Resulting in confusion or misinterpretation
  - Semantic disconnects between the original intended use and later reuse

Modeling Semantics Principle	Technical Debt Implications
16. Avoid Undertyping: Prefer specializations over generalizations where the distinction matters.	Undertyping produces ambiguity and short-circuits the modeling tool's ability to verify semantic correctness of the model. Scale: High | Propagation: High | Rework: High | Use Tax: Medium
17. Avoid Overloading: Avoid re-using the same modeling construct to represent multiple distinct concepts.	Overloading of modeling constructs produces ambiguity and it can be very difficult to determine which of those instances needs to be reworked. Scale: High | Propagation: High | Rework: High | Use Tax: Low
18. Avoid Composition Misuse: Avoid using composition when intent violates composition semantics.	Violating the semantics of composition often produces redundancy and inconsistency in a model federation. Scale: Medium | Propagation: Medium | Rework: Medium | Use Tax: Low
19. Intrinsic is Permanent: Avoid using permanent model constructs for context-dependent knowledge.	Modeling context-dependent characteristics using immutable modeling constructs results in brittle models. Scale: Medium | Propagation: High | Rework: Medium | Use Tax: Low

Model Implementation Principles and their Technical Debt Implications

- A descriptive model's implementation is fully exposed to users and its functionality is entirely dependent on users' ability to interpret and interrogate that model implementation
  - As a result, model technical debt often arises at the implementation level—not typically considered to be "architectural"

Model Implementation Principle	Technical Debt Implications
20. Don't Repeat Yourself: Minimize redundant or duplicative information in the model.	Redundant or duplicative information in the model is difficult to cleanly remove and can lead to incorrect or misleading answers to model queries. Scale: Medium | Propagation: High | Rework: High | Use Tax: Medium
21. Avoid Brittle Views: Views should be built to be readily sustained as the model grows.	Views can easily be constructed in a manner that requires significant maintenance to keep updated as the model evolves. Scale: Low | Propagation: Low | Rework: Medium | Use Tax: High
22. Cite Your Sources: Descriptive models that embody knowledge should have documented sources.	If information sources are not documented at the time their model elements are created, those sources may be difficult or impossible to find later. Scale: Low | Propagation: Low | Rework: Medium | Use Tax: Low
23. Use Units: Value properties that should have units but don't are a disaster waiting to happen.	Value properties with implied units are a source of substantial risk of misuse or misinterpretation, with difficult, highly propagative rework. Scale: Medium | Propagation: High | Rework: Medium | Use Tax: Medium
24. Build Quality In: Models should be built with quality in mind from the beginning.	Poor model quality can be difficult to refactor, can negatively impact the utility of the model to correctly answer questions, and often proliferates. Scale: High | Propagation: Medium | Rework: Medium | Use Tax: High

Summary

- The **technical debt** concept widely used in the software domain needs to be modified to the domain of descriptive models
  - Technical debt is rework that is deferred to the future for expediency
  - Model architects need to understand how principal, interest, and taxes influence model value

- The technical debt **implications** of key model architecture and implementation **decisions** need to be understood by model architects to enable the right decisions to be made
  - These decisions are made explicitly or implicitly by modelers when developing descriptive models
  - Accepting (the right) technical debt is often the right choice

- To illustrate the model technical debt concept, several key examples of modeling principles pertaining to model **purpose** and **implementation** are described along with their implications on model technical debt

Backup

2. Model Federation Principle—Consider Federation Early (CFE)

Scale: High	Propagation: High	Rework: High	Use Tax: Low

- When a model is first created, its size and stakeholders' expectations are small
  - This limits the immediacy of concerns like scaling and federation
- Accordingly, most modeling efforts begin with a single, monolithic model
  - This simplifies development and model management
- However, as the model grows in size and use, the value of federation typically grows also
  - But model federation architecture becomes increasingly difficult to change as the model grows
  - Not consciously architecting a model federation often results in a large "balloon payment" required to pay off the debt
- The taxes to both modelers and users associated with delaying federation are deceptively low
  - The agile development principle of waiting until "the last responsible moment" to make decisions often results in model architectural decisions—particularly about federation—being made too late to avoid significant rework costs
- Modeling projects are generally better off addressing federation proactively, as early as possible
  - However, this doesn't mean all model projects should use federation
  - Federation should be done to address specific functional objectives for the models
  - These model objectives should be key drivers for the decisions about how to structure the federation

Model federation architecture becomes increasingly difficult to change as the model grows in size and use

3. Model Federation Principle—Partition for Cohesion (PfC)

Scale: High	Propagation: High	Rework: High	Use Tax: High

- A key role of the architect is partitioning a large component into discrete components
- In the software domain, this principle is essentially a combination of two principles
  - The Reuse/Release Equivalence Principle: “the granule of reuse is the granule of release,” and
  - The Common Reuse Principle: “the classes in a component are reused together” (Martin & Martin, 2006)

- Partitioning model scope into individual federated models should be driven by (often competing) considerations of model governance and model usage

- Aligning model partitions with organizational responsibilities for model content improves the efficiency of model governance by easing the burden of formal coordination
- Model usage is facilitated when fewer model boundaries are crossed when executing queries
  - Each boundary crossing can result in semantic mismatch, complicate query construction, or impede tool performance
  - Efficiency is maximized when the most frequent model usages stay within a model’s boundary

- The impact of misalignment—too much coupling, not enough cohesion—can manifest in substantial additional effort needed to build, sustain, and use those models

Martin, R. & Martin, M. (2006). Agile Principles, Patterns, and Practices in C#. Pearson.

Misalignment with content boundaries complicates model governance and sustainment

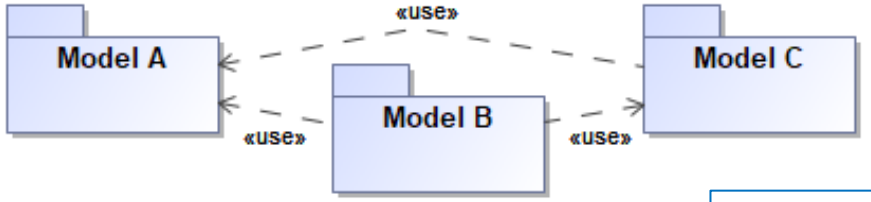
Misalignment with usage boundaries complicates model use

4. Model Federation Principle—Federate to Interrogate (FtI)

Scale: High	Propagation: High	Rework: High	Use Tax: High

- Model federation is the establishment of directional model usage relationships between independently managed models
- Those directional relationships should be designed to avoid model usage cycles or interdependencies
  - These can result in performance problems as models are loaded and reloaded, potentially indefinitely
- In software, this principle is an extension of the Acyclic Dependencies Principle
  - Which states: “Allow no cycles in the component dependency graph” (Martin & Martin, 2006).
- The diagram on the left depicts a simple dependency cycle, in which models A and C are interdependent
  - Federations should be architected to avoid this but also ensure usage relationships are aligned with the most important queries
- Queries navigating the reverse paths can be accommodated by introducing an additional “bridging” model (Model B) that uses both of the dependent models as shown in the diagram on the right

1) a model usage cycle



2) Breaking the cycle via a “bridging” model

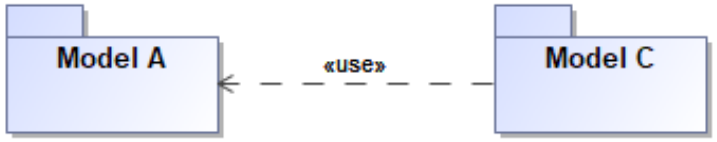
Martin, R. & Martin, M. (2006). Agile Principles, Patterns, and Practices in C#. Pearson.

Query navigation paths opposing model usage paths complicate using the model to answer questions

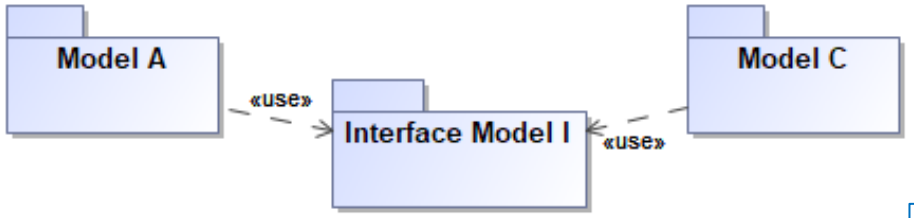
5. Model Federation Principle—Dependency Inversion Principle (DIP)

Scale: High	Propagation: High	Rework: High	Use Tax: High

- This is analogous to the OO software principle of the same name, which specifies that more abstract software modules should depend on less abstract software modules (Martin & Martin, 2006)
  - More concrete modules are typically more volatile than more abstract modules
  - Dependency management is facilitated when dependencies are more stable than modules that depend on them
- In the diagram on the left, the directionality of dependency between Models A and C is from the more concrete C to the more abstract A, consistent with DIP
  - This direction of the dependency results in much less frequent model churn at the interface
  - The more concrete implementation model is generally much more frequently updated than the more abstract model
- In the diagram on the right, an interface model I can be used to connect Models A and C
  - This insulates each model from volatility in the other, since model I is more abstract and more stable than A and C



1) Concrete model C depends on abstract model A



2) Interface model is more stable than models A and B

Martin, R. & Martin, M. (2006). Agile Principles, Patterns, and Practices in C#. Pearson.

More concrete models typically more frequent and deeper change, impacting models that depend on those models

6. Model Federation Principle—Define Clear Interfaces (DCI)

Scale: High	Propagation: High	Rework: High	Use Tax: High

- Whenever models are federated, that federation is implemented by directly connecting elements in one model with elements of another model using relationships
- Connections between models should be well-defined to establish both:
  - A consistent mechanism for semantically connecting those concepts represented by those model elements
  - A consistent mechanism for querying those models.
- Specific model elements in model A should be connected to specific model elements in model B with specific relationships carrying specific semantic meanings
- It is particularly important to define those model interfaces when crossing modeling language boundaries

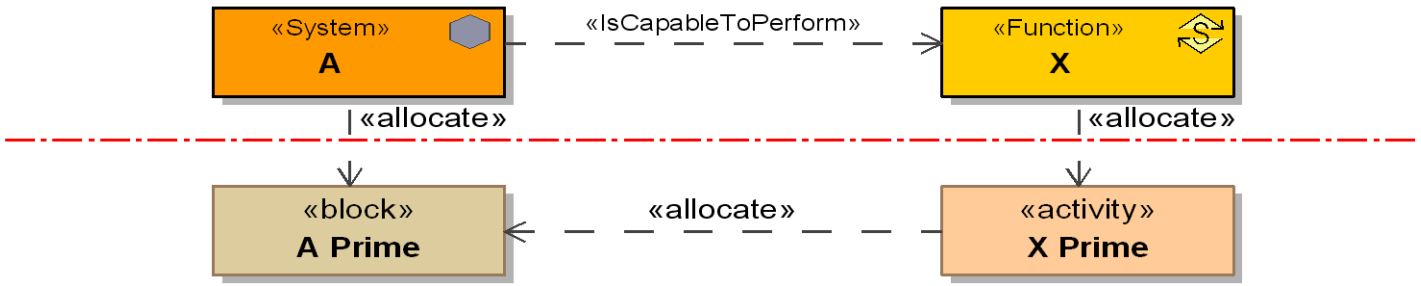


Figure adapted from:
Martin, J.N. & Brookshier, D. (2023). Linking UAF and SysML Models: Achieving Alignment between Enterprise and System Architectures. 33rd INCOSE International Symposium.

Ill-defined model interfaces often results in inconsistent inter-model connections and encapsulation breaking

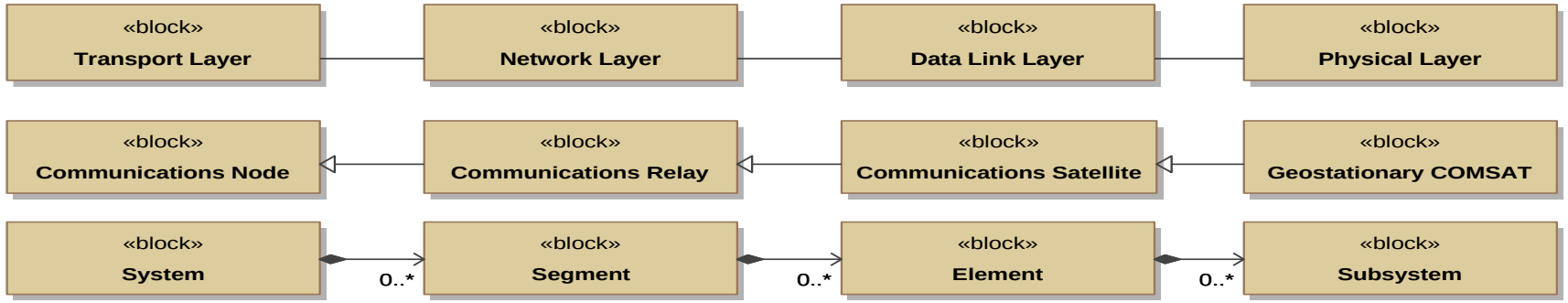
7. Model Layers Principle—Just Enough Layers (JEL)

Scale: Medium	Propagation: High	Rework: High	Use Tax: Low

- When defining abstraction layers, it is often valuable to create multiple logical layers or physical layers of abstraction to adequately address architectural tradeoffs in the logical or physical trade space
  - It can be difficult to know a priori how many abstraction layers will be appropriate for a modeling application

- Making the wrong choice leads to greater technical debt
  - Too few layers elevates the risk of needing to add additional intermediate layers later, with substantial rework cost
  - Too many layers drives additional complexity and taxes to model builders and users

- Similarly, taxonomic levels—generalization and composition hierarchies—should be defined cautiously
  - Too few layers can create difficult rework for each specialization of each model element
  - Too many layers can lead to an explosion of additional model elements



Too few layers produce technical debt that must be repaid when layers must be added later

Too many layers add extra complexity to the model

8. Model Layers Principle—Don't Cross Streams (DCS)

Scale: Medium	Propagation: High	Rework: High	Use Tax: Low

- Modeling multiple levels of abstraction or taxonomy is usually highly beneficial for most MBSE application
- Care must also be taken to avoid creating too many different connections between those levels
  - Particularly when this results in the violations of the context of the abstraction
- Mixing structure and abstraction in a single hierarchy or combining contextual tenses within the same model are two often-seen examples
  - Easily render the model inconsistent, incoherent, and very difficult to correct
- The modeling methodology should clearly define the distinction between the layers and identify specific interface points between those layers to enforce consistency and avoid breaking encapsulation
  - Communication across abstraction layers dilutes the separation of concerns those abstraction layers were intended to provide.

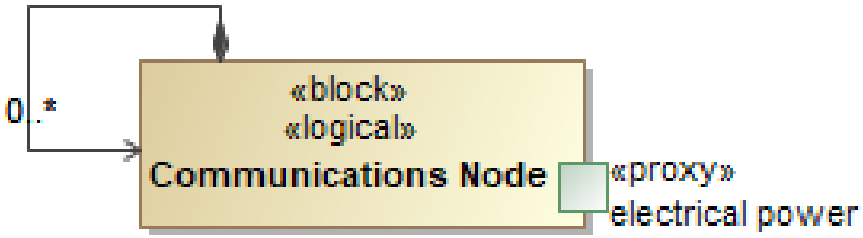
Mixing abstraction and taxonomic levels in the same hierarchy, or uncontrolled connections between levels, results in difficult use and painful rework

9. Model Layers Principle—Scale Free Decomposition (SFD)

Scale: Medium	Propagation: High	Rework: Medium	Use Tax: Medium

- When creating hierarchical structures, modelers often overuse bespoke levels of decomposition
  - Often by creating a series of base classifiers or stereotypes that represent a specific taxonomy of distinct element types that may not be warranted

- In many situations, the natural representation would be a tree structure in which the distinction between single elements of the tree and composite elements of the tree is irrelevant
  - e.g., a scale-free mode of decomposition using recursion
    - This is an efficient mechanism for capturing this structure
      - Minimizes the addition of extraneous decomposition layers and the elements, relationships, and other complexities associated with those additional layers
  - This approach also facilitates satisfaction of the other two Model Layers Principles
    - Just Enough Layers
    - Don’t Cross Streams



Defining a specific hierarchy of levels often adds unnecessary complexity and can be overly constraining

10. Modeling Domain Principles—Other People's Profiles (OPP)

Scale: High	Propagation: High	Rework: High	Use Tax: High

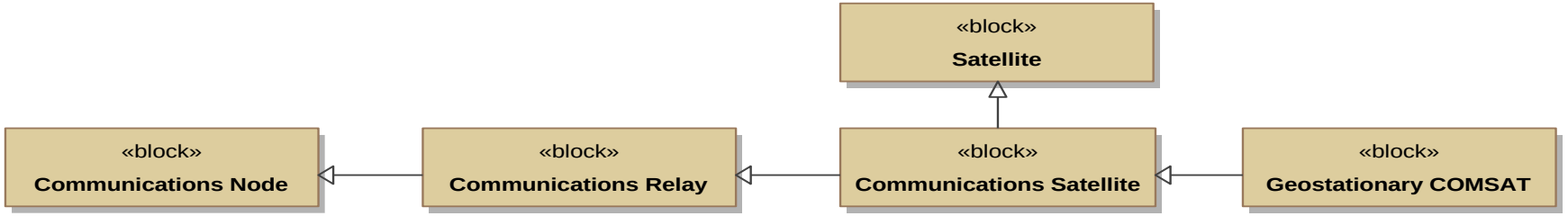
- One of the most painful experiences for a model reviewer is to behold the inadvertent reinvention of the square wheel
  - Modelers often eschew widely used standards for representing well-known concepts in lieu of creating their own unique variation
- While it can be very useful to create new metamodels and profiles to better represent vital concepts within the model's domain, this also has a downside
  - Often, these homegrown approaches are narrowly focused and poorly documented, reducing their reusability and understandability.
- Furthermore, unique profiles must be continually maintained through the life of the project
  - Often that maintenance cost can be eliminated or significantly reduced by leveraging standards
- By converging on a preferred set of these standards and contributing to their evolution, the MBSE community can better leverage reuse, improve model interoperability, and more efficiently use their resources

Reinventing the wheel demands continued maintenance investment and reduces interoperability

11. Modeling Domain Principles—Single Responsibility Principle (SRP)

Scale: Medium	Propagation: Medium	Rework: Medium	Use Tax: Low

- Definitions should be minimal in scope
  - i.e., should represent a single coherent concept rather than a combination of multiple distinct concepts
- Such a composite concept is usually best represented in the model by using multiple inheritance
  - i.e., by multiple generalization relationships to those relevant parent concepts
- Failure to adhere to this principle often results in the need to eventually break up that definition into its constituent components
  - And deal with the substantial propagation of rework to the specializations and usages of that definition as well as users of those usages
- Modelers very experienced with OO programming can be subconsciously biased against multiple inheritance due to their software experience



Refactoring to break up combination concepts into more primitive concepts can be time consuming and error prone

12. Modeling Domain Principles—Open/Closed Principle (OCP)

Scale: Medium	Propagation: Medium	Rework: Medium	Use Tax: Low

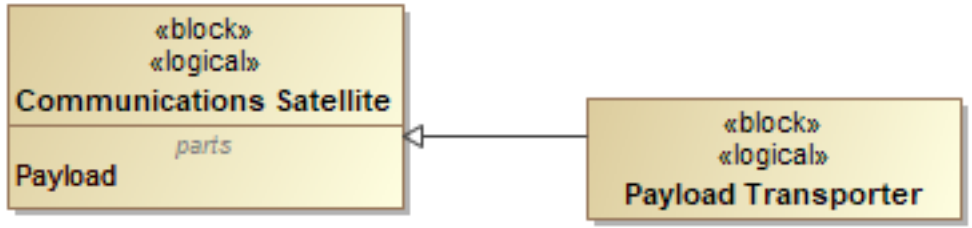
- Model element type definitions should be designed to be specialized without needing to be modified
- Typically, the modification is needed when an overly constrained conceptual understanding of the decomposition of an element is envisioned when it is first created
- While it isn't possible to anticipate all future contexts, enabling future flexibility is not difficult when done early
- In the figure, the Communications Satellite block has a Payload part with the default multiplicity of 1
  - This multiplicity limits its use in the frequently observed cases in which the satellite bears multiple payloads
  - If this block is specialized many times before the multiplicity is updated to be less constraining, this drives rework to many of its specializations
  - That rework is difficult as it typically requires manual inspection of each specialization to determine if rework is warranted

Overconstrained base classes need to be reworked to be applicable to new contexts

13. Modeling Domain Principles—Liskov Substitution Principle (LSP)

Scale: Medium	Propagation: Medium	Rework: Medium	Use Tax: Medium

- The semantics of generalization and specialization in descriptive modeling is that any specialization of a base classifier should be a valid substitute for that base classifier (Liskov, 1988; Martin, 1996)
- In descriptive models, generalization should be reserved for those contexts for which that substitution is valid, and not just used as a convenient mechanism for reuse
- In OO software, inheritance is eschewed as most use cases for inheritance are better implemented by composition
- However, in OO modeling, inheritance carries semantics of substitutability and should be reserved for that purpose



Liskov, B. (1988). Data abstraction and hierarchy. SIGPLAN Notices 23.
Martin, R. (1996). The Liskov Substitution Principle. C++ Report, Vol 9 (2).

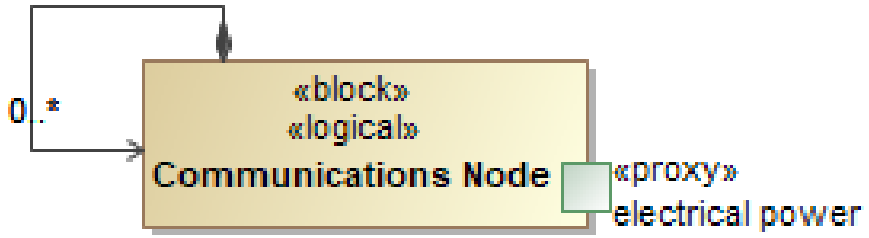
- In the diagram above, the Payload Transporter block specializes the Communications Satellite block, presumably to reuse the Payload part and other properties
  - However, it is unclear that the Payload Transporter is a suitable substitute for a Communications Satellite in any context

Violating LSP often results in confusion in the intent and proper use of both generalizations and specializations and short-circuits benefits of OO

14. Modeling Domain Principles—Interface Segregation Principle (ISP)

Scale: Medium	Propagation: Medium	Rework: Medium	Use Tax: Low

- Type definitions should avoid defining features that are not required (or even meaningful) in all of its specializations
- In the diagram below, the Communications Node has an electrical power port whose multiplicity value makes it non-optional
  - As a result, all Communications Nodes must have one, even if this would not make sense in reality
    - The node might be internally powered and entirely self-contained
    - The port may represent a level of abstraction for which electrical power connections are not appropriately modeled
  - The extraneous port then becomes both an encumbrance to continued maintenance and a potential source of confusion
  - It is often difficult to deactivate those features for those specializations that don't use them, particularly if the model containing the original type definition is not available for editing



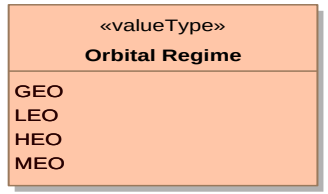
Violating ISP often introduces violations of other principles (like LSP) as workarounds and deactivating features can be difficult to implement

15. Modeling Domain Principles—Reify for Reuse (RfR)

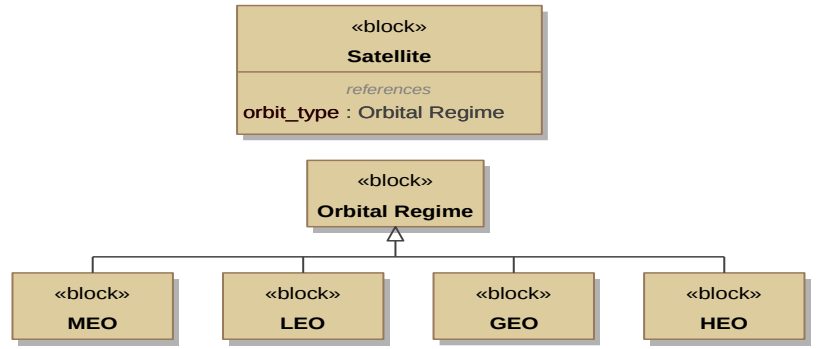
Scale: Medium	Propagation: High	Rework: High	Use Tax: Medium

- Modelers often rely too heavily on primitive types to represent items that are meant to be heavily reused
  - This results in many opportunities for failure, e.g., misspellings or alternative word choices
- Enumerations are generally a better choice
  - Prevents misspellings or alternative word choices since the potential values are constrained
- However, if the concepts represented by the value are frequently reused in different contexts it is often better to reify the concept to represent it
  - i.e., create a class to represent it—most often a block in SysML
- This approach provides greater flexibility to model users and facilitates model maintenance and evolution
  - The concept appears only once in the model rather than countless times buried within other model elements
  - Using a model element rather than an enumeration offers opportunities to take advantage of generalization

String primitive



Enumeration



Reification

Avoiding reification often results in accumulation of more primitive types, which are more prone to typographic errors and impede model queries

16. Modeling Semantics Principle—Avoid Undertyping

Scale: High	Propagation: High	Rework: High	Use Tax: Medium

- Modelers sometimes underspecify model elements
  - Often preferring to avoid being constrained by deciding on a narrowly defined type or stereotype
- An example of this is shown in Figure 1
  - The single stereotype «Protocol Link» is used to type connections regardless of whether they are appropriately connected at that level of abstraction

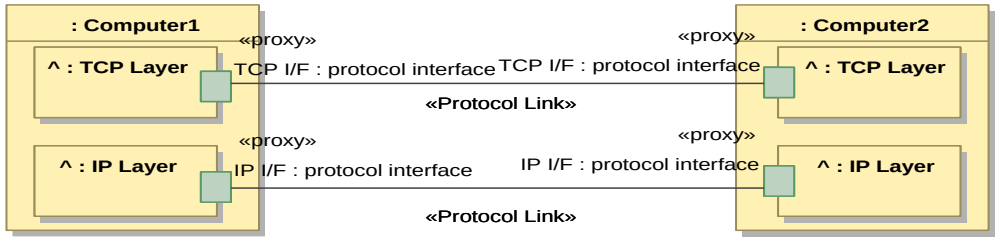


Figure 1: Example of undertyping

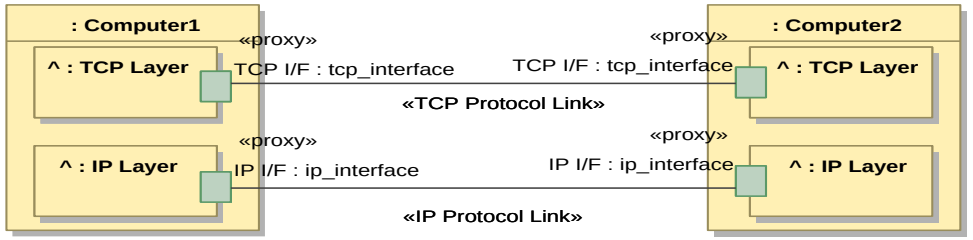


Figure 2: Example without undertyping

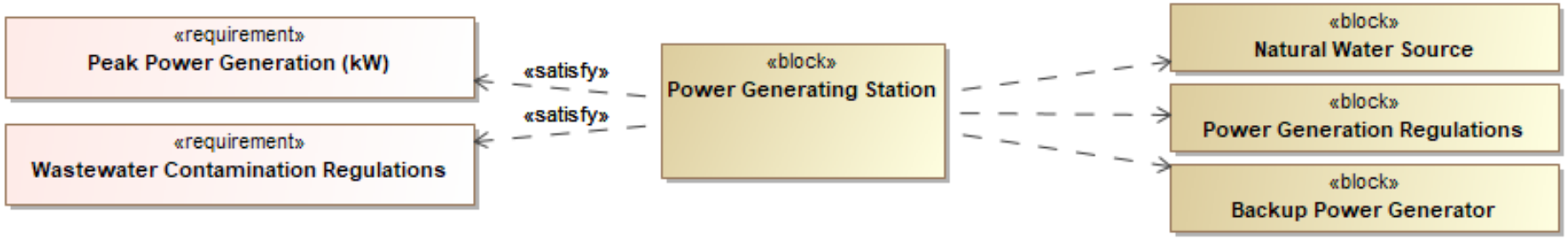
- Often, a suitable corrective action is to create a set of specializations of the type or stereotype to use in these different roles, as shown in Figure 2
  - Here, «TCP Protocol Link» and «IP Protocol Link» are specializations of «Protocol Link»
  - This allows one to treat the different types or stereotypes as the same or different, depending on the context of the question being addressed

Undertyping produces ambiguity and short-circuits the tool's ability to verify semantic correctness of the model

17. Modeling Semantics Principle—Avoid Overloading

Scale: High	Propagation: High	Rework: High	Use Tax: Low

- Modelers often overload concepts with multiple meanings
  - Often this results from relying only on standard concepts defined by the language and not extending the language to include new concepts to make the distinction concrete



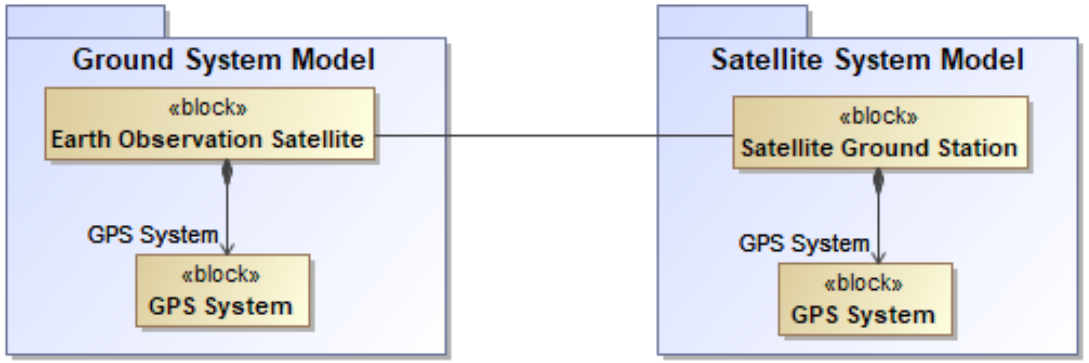
- In this example of semantic overloading the «dependency» and «satisfy» relationships are each used in the same model to express multiple distinct meanings
  - The three «dependency» relationships each represent very different types of dependencies
  - The two «satisfy» relationships each represent very different notions of requirement “satisfaction”
- Overloading results in rework that can be very difficult to find since each instance must be assessed separately to determine which of the overloaded meanings is the correct one
- This model view shown is not necessarily *wrong*, but overloading adds some risk of technical debt that more refinement will be needed in the future involving major rework

Overloading of modeling constructs produces ambiguity and difficulty in identifying needed rework

18. Modeling Semantics Principle—Avoid Composition Misuse

Scale: Medium	Propagation: Medium	Rework: Medium	Use Tax: Low

- Modelers often overuse composition when describing entities that are not intended to be duplicated
  - e.g., when block definition B1 has a composition relationship to block definition B2, this relationship establishes the existence of an individual usage b2 of block definition B2 in the context of B1
  - That individual usage is distinct and separate from every other usage
  - While this is often appropriate, in other cases, it is dangerous and misleading
    - In the diagram below, the two model elements representing the GPS system are intended to represent the same system, not two separate copies of that system
- The problem often doesn't manifest when the models are used in isolation but emerges when the models are federated
- In this example, the semantics of composition inhibit the proper interpretation of a single GPS system within that federation
  - Replacing the composition relationship with a reference association would enable the same GPS System to be represented in both models by the same usage



Violating the semantics of composition often produces redundancy and inconsistency in a model federation

19. Modeling Semantics Principle—Intrinsic is Permanent

Scale: Medium	Propagation: High	Rework: Medium	Use Tax: Low

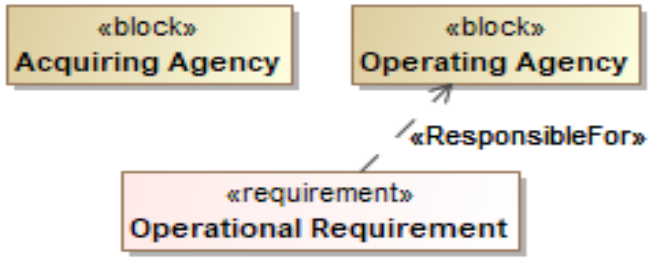
- Modelers often use permanent modeling constructs to represent characteristics that are transitory or context-dependent
  - This results in ambiguity or misinterpretation when the context changes
- Commonly seen examples of these include the specification of stereotypes to represent the concept of a “system of interest” or a “stakeholder”
  - Both of these concepts are highly context-dependent and are not intrinsic properties of the entity being modeled
- While this ambiguity is itself undesirable, the larger danger is not simply misinterpretation of that specific model element, but the propagation of the underlying assumption throughout the model
  - This can significantly hinder the model’s reuse and interpretation within a model federation
- Instead, contextually dependent characteristics should be modeled using modeling constructs that properly convey the context in which the assertion is being made
  - e.g., the characteristic of being a stakeholder is more appropriately modeled as a role or a relationship
  - The nature of that relationship may differ significantly between stakeholders

Modeling context-dependent characteristics using immutable modeling constructs results in brittle models

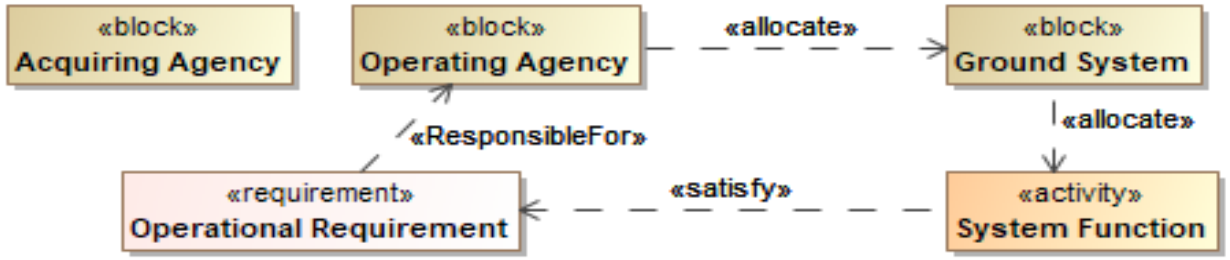
20. Model Implementation Principle—Don't Repeat Yourself

Scale: Medium	Propagation: High	Rework: High	Use Tax: Medium

- MBSE facilitates consistency management by enabling more efficient reuse of knowledge
- While most modelers are able to prevent obvious duplication ("copy-paste") of information within models, a more subtle form of information duplication is often much more difficult to prevent
  - In the early stages of a model, it is common to implement a relationship that seems intuitive or obvious
  - Over time, the model grows to include additional direct and indirect relationships
  - Many of these relationships—particularly the indirect ones—are essentially redundant



a) direct relationship in early model



b) multiple direct and indirect relationships in refined model

- This is not necessarily an error, as both relationships may be legitimate
  - However, refactoring the model to remove duplications and inconsistencies is time-consuming and error prone, particularly since these indirect relationships often grow exponentially over time

Duplication and redundancy is a common technical debt incurred in descriptive models

21. Model Implementation Principle— Avoid Brittle Views

Scale: Low	Propagation: Low	Rework: Medium	Use Tax: High

- One benefit of MBSE is that model views can be automatically kept up to date as the model evolves
  - Changes made to an element in the model can be immediately reflected everywhere that element appears
- While this is typically true, it is important to note that in graphical views this only occurs where that element already appears
  - A newly added model element is rarely automatically added to an existing diagram
- Model views often carry hidden or explicit assumptions that may not be enforced by the model view specification
  - e.g., a graphical model view may be intended to show all of the requirements that have been allocated to a given system component; if a qualifying requirement is later added to the model, the graphical model view is generally not automatically updated to reflect that change, so the model view is no longer consistent with its intent
- Brittle views require substantial maintenance effort to keep up to date as the model evolves
  - When not kept up to date, users can be misled by these views that are no longer consistent with their intent
- Dynamically generated views tend to be less prone to this problem
  - Relation maps, dependency matrices, tables, etc. are regenerated from the model on demand
  - However, poor choices of scope can result in these views not being regenerated consistently
    - This is commonly seen when modelers prefer to use package containment to avoid creating new stereotypes

Brittle views require substantial recurring maintenance lest they mislead users

22. Model Implementation Principle— Cite Your Sources

Scale: Low	Propagation: Low	Rework: Medium	Use Tax: Low

- In MBSE, descriptive models replace documents as the embodiment of SE information
  - However, the content of those models often originates from other sources, e.g., documents, analyses, or other, disconnected models
- Establishing the provenance of this data is often essential to provide users with confidence in the accuracy of the representation of this information in a model
- When information sources are not documented at the time their model elements are created, those sources can often be difficult or impossible to find later
- Furthermore, when those information sources are updated, it can be difficult to ensure the appropriate updates are made everywhere needed in the model
  - Reifying reference sources—creating specific model elements representing them rather than relying on free-form text strings to document those sources—can be helpful
  - A robust configuration management approach can facilitate keeping the model synchronized with those sources

Failing to document sources can result in untrustworthy models

23. Model Implementation Principle— Use Units

Scale: Medium	Propagation: High	Rework: Medium	Use Tax: Medium

- SysML 1.4 introduced a mature and comprehensive standard approach for assigning units to value properties
  - However, it is not as widely used as it should be

- Value properties that should have units but do not are at a substantial risk for being misused or misinterpreted with potentially catastrophic consequences

- Missing units can often propagate exponentially through the model
  - Resulting in substantial rework to find and correct not directly derived properties, opaque expressions, and other indirect impacts

Poor discipline with units can result in substantial technical debt to correct

24. Model Implementation Principle— Build Quality In

Scale: High	Propagation: Medium	Rework: Medium	Use Tax: High

- Quality is often much more critical in descriptive models than in software
  - Since users are exposed to the model's implementation details, not just abstracted behaviors
- Poor attention to model quality during construction often results in proliferation and propagation of defects
  - Increases the difficulty of rework
  - Also impacts users' ability to create queries and trust that the results are complete and correct
- Automated model checking, modeler training, and peer reviews are vitally important tools
  - To strengthen model quality and detect errors in either/both the model or the design of the entity being modeled
- Disciplined attention to model quality during construction is critical to minimize contagion and facilitate trust in models
- Failure to ensure model quality can lead to stakeholders losing faith and confidence in the models to serve as the authoritative source of truth of SE knowledge
  - This can threaten the continued viability of the effort

Poor model quality can be difficult to refactor and reduces utility of and confidence in the model