Reverse Engineering Product Lines in Agile Environments

Jane Cleland-Huang, PhD

Software Product Line Conference, Seville, Spain 2017

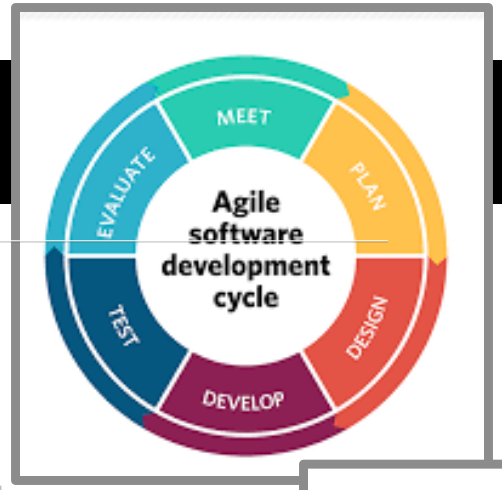
JaneClelandHuang@nd.edu

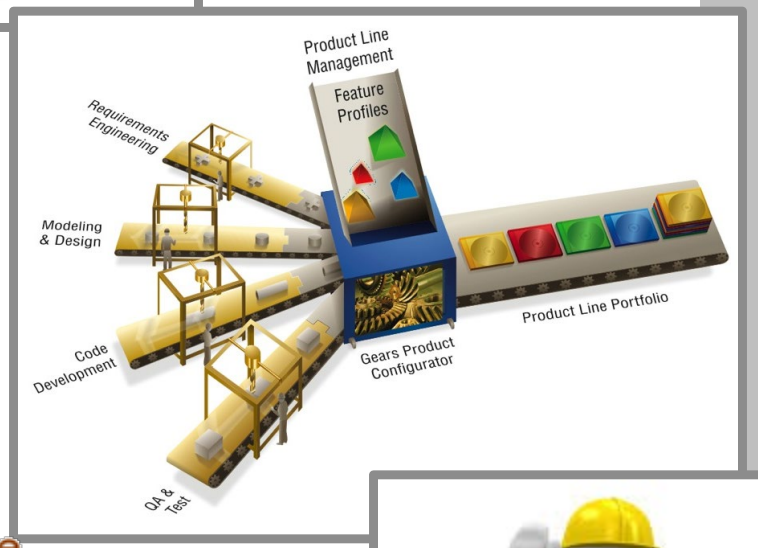
Department of Computer Science and Engineering

University of Notre Dame

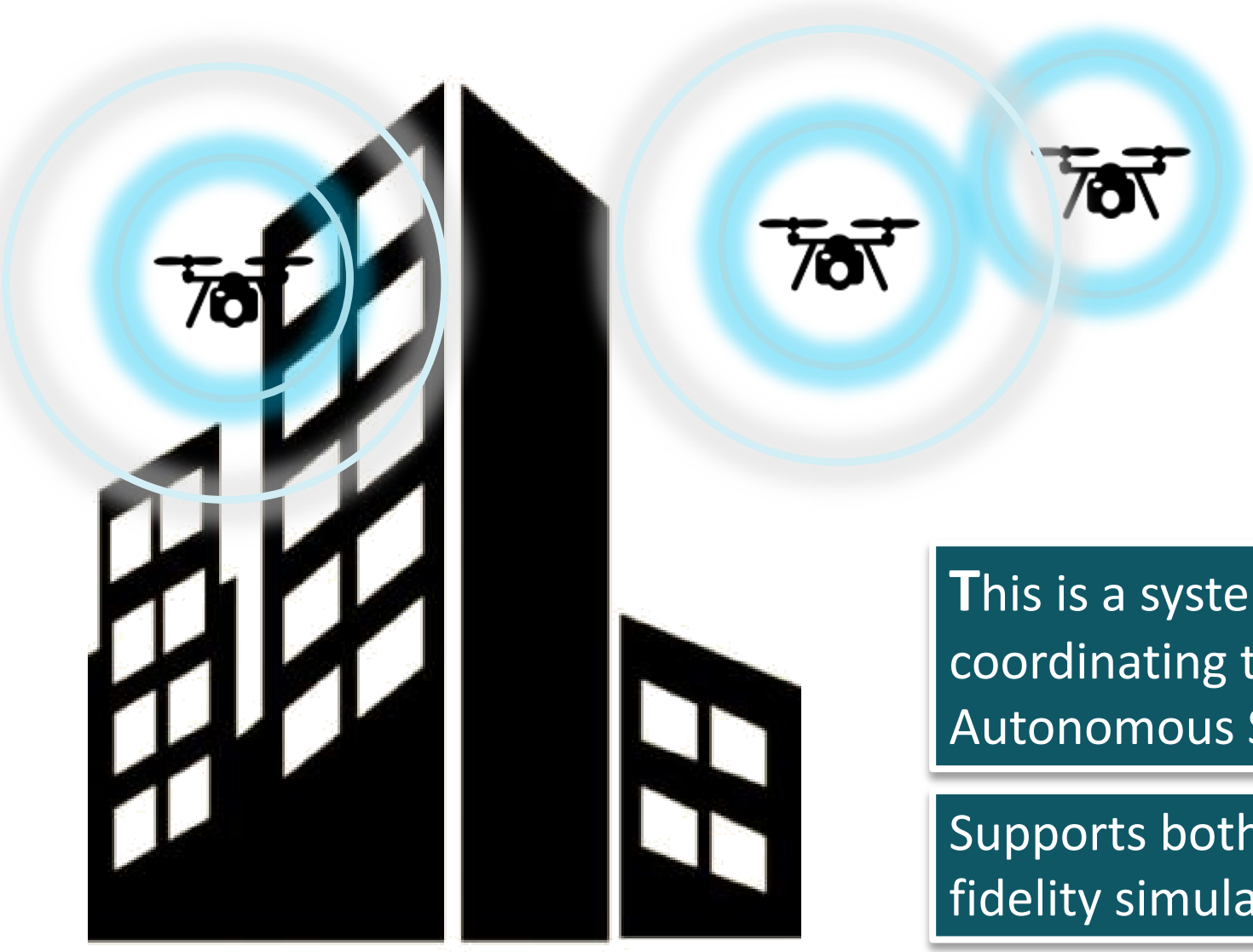
Some of the work described in this talk was funded by the US National Science Foundation under Grants CCF-0959924 and CCF-1265178.

What I'm going to talk about..





Dronology: Our Context and Challenge

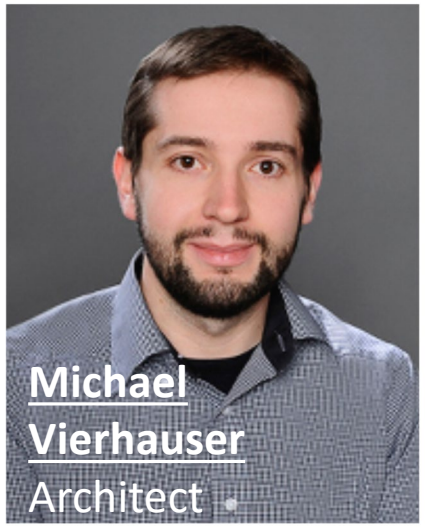


Focus on constructing and releasing a real-world, evolving safety-critical development environment to support Requirements Engineering research

This is a system for controlling and coordinating the flights of Unmanned Autonomous Systems in an urban airspace

Supports both physical UAS and high-fidelity simulators

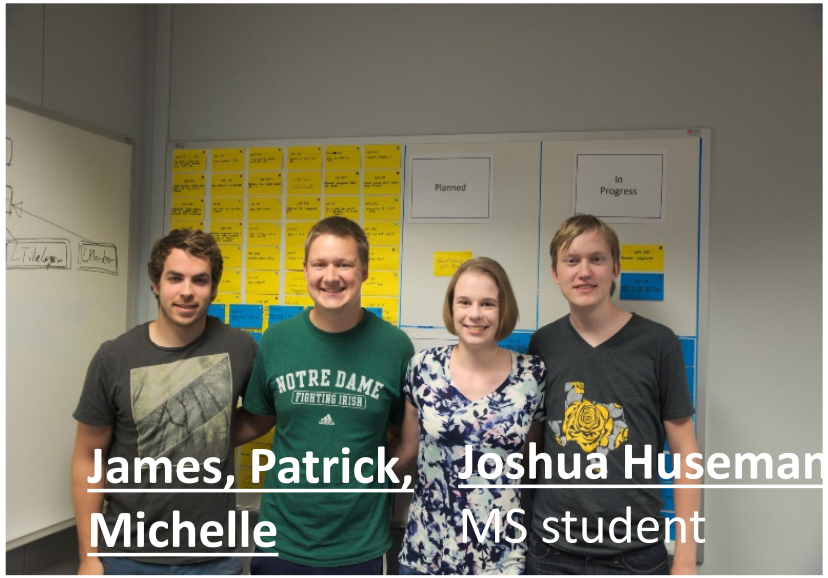
Team Work



Michael Vierhauser
Architect

Jane Wyngaard
Hardware

Jinghui Cheng
Post Doc / UI



James, Patrick, Michelle
Joshua Huseman
MS student

Sean Bayley
PhD Student

Alex Madey
Undergraduate

Quinlan McMillan
Undergraduate

Greg Madey
Frank

Dronology Stakeholders

http:/Dronology.info




You?

<u>Requirements Engineering researchers</u> interested in using Dronology to support research into traceability, forensic requirements, goal modeling, runtime adaptation….

<u>End users</u> interested in deploying Dronology to support specific scenarios of use.

Students

General public

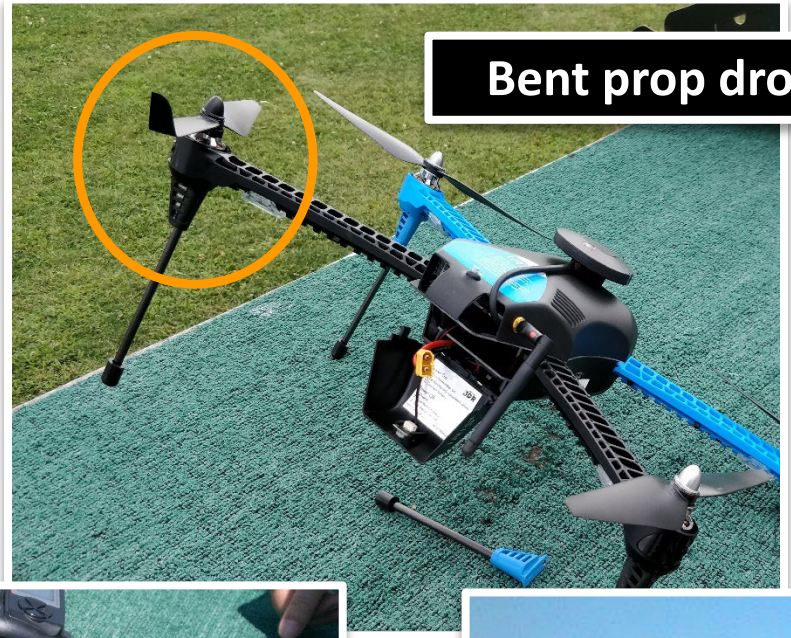
Funders

Stakeholders' needs ultimately drive feature prioritization and variability points.

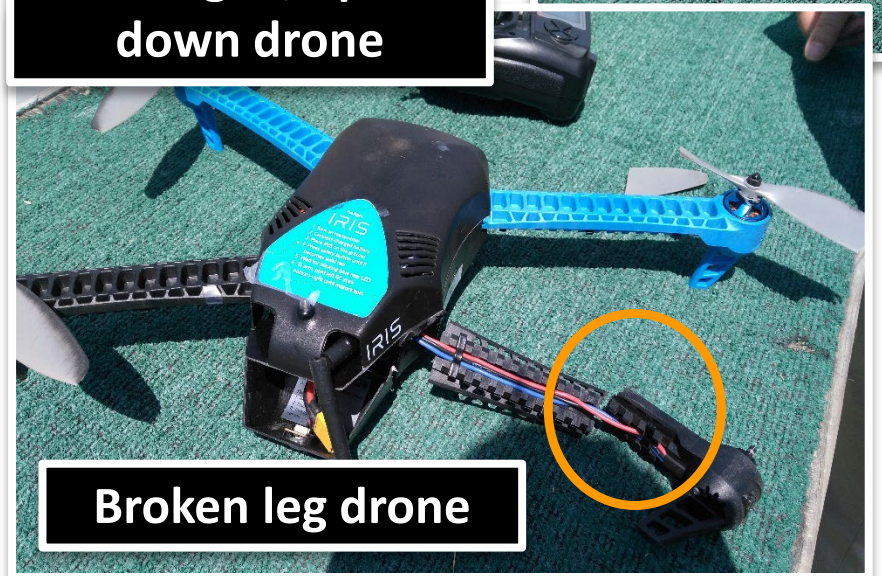
Dronology: Our Context and Challenge

Dronology: Our Crashes

Trajectory challenged, upside down drone

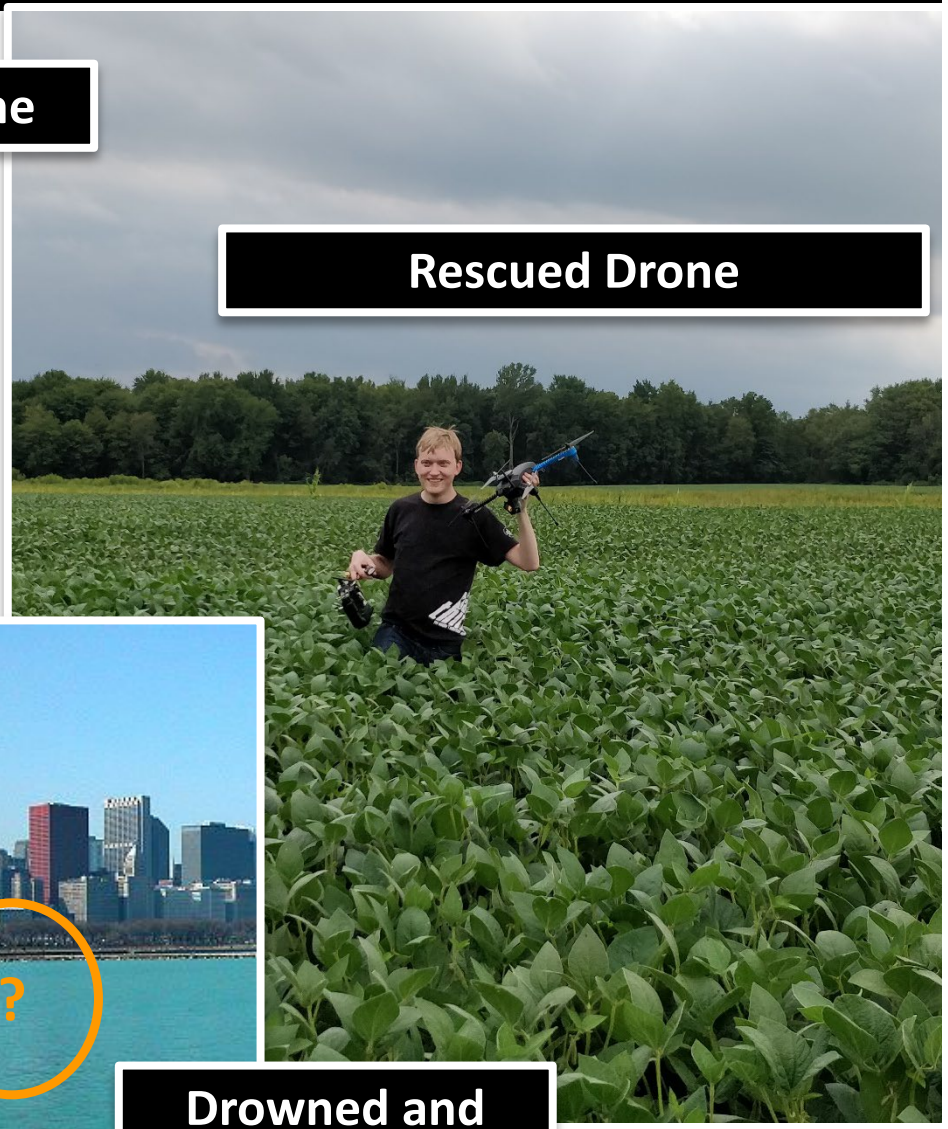


Bent prop drone



Broken leg drone

Drowned and missing drone



Rescued Drone

It could be worse..

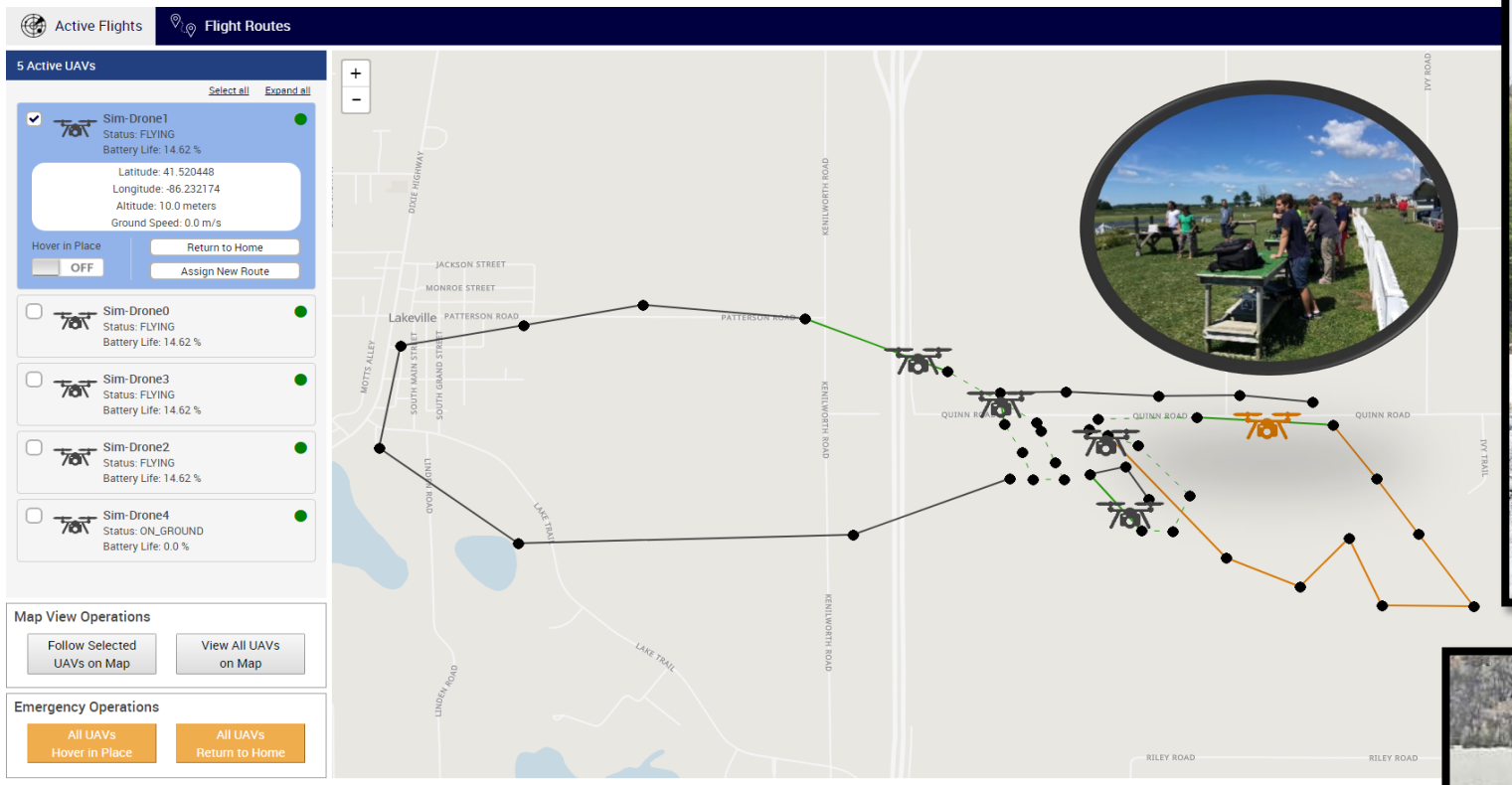
It could be worse..

Or potentially damaging

Most of the time..

Dronology Project @ Notre Dame

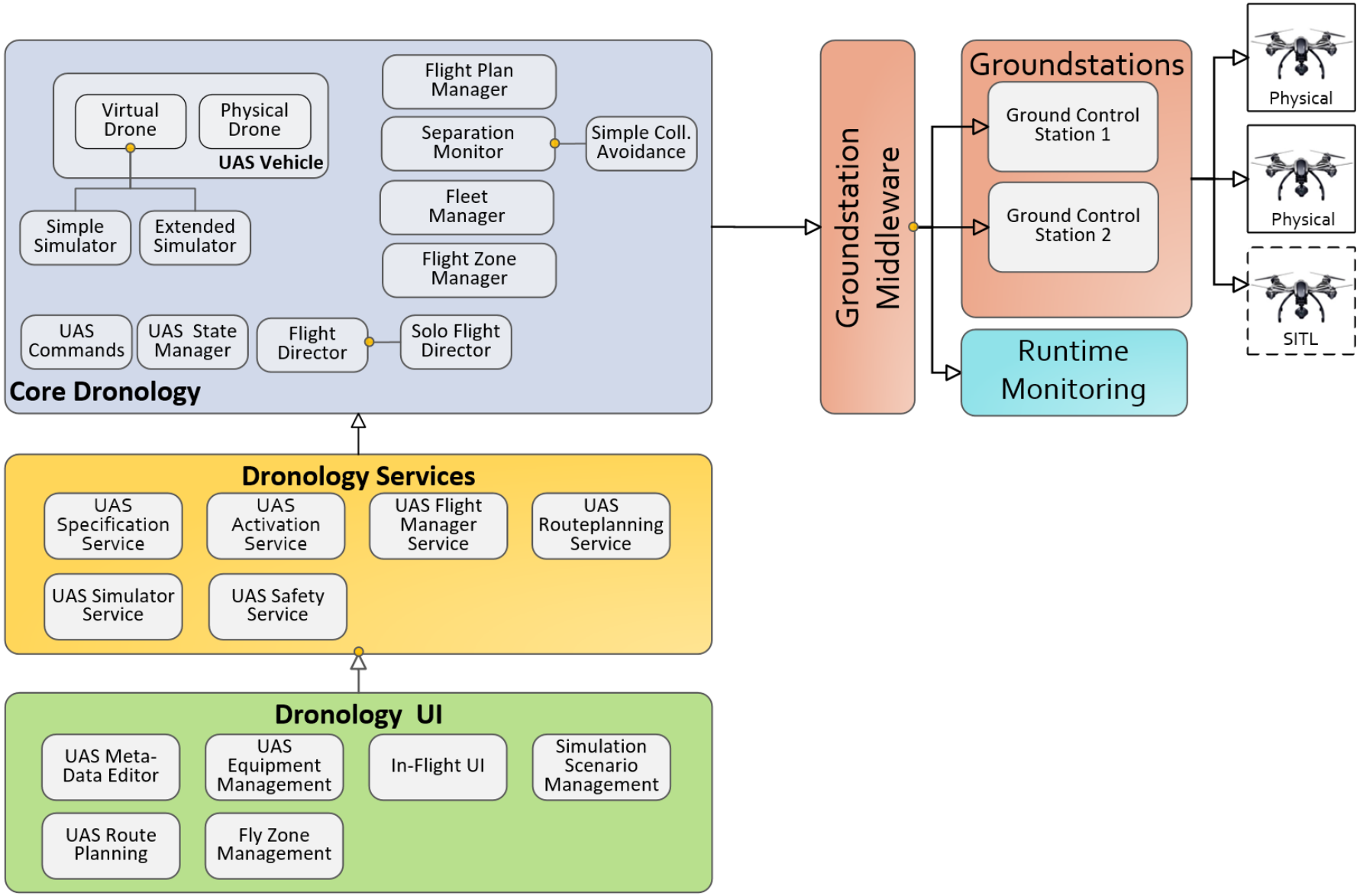
http://sarec.nd.edu/pages/Dronology.html



A platform for coordinating the flight of UAVs. Supports research in safety assurance, runtime monitoring, & adaptation.

Developed by the Notre Dame Team: Michael Vierhauser, Jane Wyngaard, Jinghui Cheng, Sean Bayley, Greg Madey, Joshua Huseman, Jane Cleland-Huang, & more…

Dronology Architecture



Where two worlds meet..

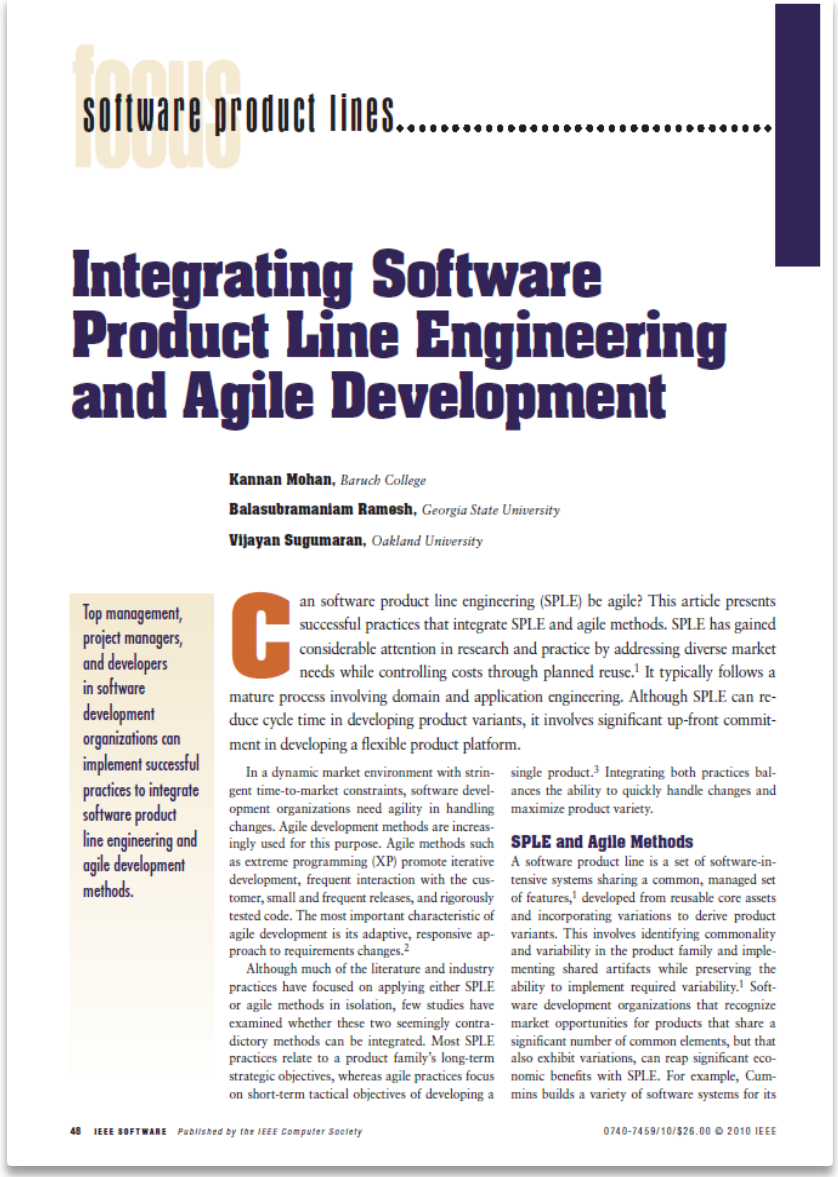
A Product Line

A portfolio of related products

Agile Development

A lightweight software process centered around iterative development

Can we merge them?



focus software product lines

Integrating Software Product Line Engineering and Agile Development

Kannan Mohan, *Baruch College*
Balasubramaniam Ramesh, *Georgia State University*
Vijayan Sugumaran, *Oakland University*

Top management, project managers, and developers in software development organizations can implement successful practices to integrate software product line engineering and agile development methods.

Can software product line engineering (SPLE) be agile? This article presents successful practices that integrate SPLE and agile methods. SPLE has gained considerable attention in research and practice by addressing diverse market needs while controlling costs through planned reuse.[1] It typically follows a mature process involving domain and application engineering. Although SPLE can reduce cycle time in developing product variants, it involves significant up-front commitment in developing a flexible product platform.

In a dynamic market environment with stringent time-to-market constraints, software development organizations need agility in handling changes. Agile development methods are increasingly used for this purpose. Agile methods such as extreme programming (XP) promote iterative development, frequent interaction with the customer, small and frequent releases, and rigorously tested code. The most important characteristic of agile development is its adaptive, responsive approach to requirements changes.[2]

Although much of the literature and industry practices have focused on applying either SPLE or agile methods in isolation, few studies have examined whether these two seemingly contradictory methods can be integrated. Most SPLE practices relate to a product family's long-term strategic objectives, whereas agile practices focus on short-term tactical objectives of developing a single product.[3] Integrating both practices balances the ability to quickly handle changes and maximize product variety.

SPLE and Agile Methods

A software product line is a set of software-intensive systems sharing a common, managed set of features,[1] developed from reusable core assets and incorporating variations to derive product variants. This involves identifying commonality and variability in the product family and implementing shared artifacts while preserving the ability to implement required variability.[1] Software development organizations that recognize market opportunities for products that share a significant number of common elements, but that also exhibit variations, can reap significant economic benefits with SPLE. For example, Cummins builds a variety of software systems for its

48 IEEE SOFTWARE Published by the IEEE Computer Society 0740-7459/10/$26.00 © 2010 IEEE

Software Product Line Engineering

- Upfront planning
- Predict commonalities & variabilities
- Mature processes
- Early architecture design
- Platform adaptability to accommodate future variabilities
- Formal documentation

Agile Development

- Just enough planning
- Requirements (user stories) needed now
- No futuristic designs
- Face-to-face informal communication

Both aim to handle changing requirements

What do others say?

To be presented at 26th Annual INCOSE International Symposium (IS2016)
Edinburgh, July 18-21, 2016

The Best of Both Worlds: Agile Development Meets Product Line Engineering at Lockheed Martin

Susan P. Gregg, Rick Scharadin
Lockheed Martin
{susan.p.gregg, richard.w.scharadin}@lmco.com

Paul C. Clements
BigLever Software
pclements@biglever.com

Abstract. Agile development has long been touted as way to optimize software development team efficiency and improve project success. Product line engineering (PLE) brings large-scale improvements in cost, time to market, product quality, and more. Can these two paradigms work in concert with each other? This paper details the experience of Lockheed Martin as it introduced large-scale agile development practices on one of its largest and most successful product line engineering efforts.

Introduction

Agile software development refers to a group of software development methods in which requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and encourages rapid and flexible response to change [8]. Its adherents tout higher quality systems, delivered faster, which much better match customer needs and expectations.

Systems and software product line engineering, or "product line engineering (PLE)" for short, is a way to engineer a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. Considering a portfolio as a single entity to be managed, as opposed to a multitude of separate products to be managed, brings enormous efficiencies in production and maintenance; these efficiencies are delivering order-of-magnitude improvements in engineering cost, time to market, staff productivity, product line scalability, and quality [9].

What happens when an organization tries to apply both of these groundbreaking, organization-changing methodologies at the same time? Can they work together at all? Is PLE, which relies on cross-product planning and well-entrenched coordination, compatible with Agile, the very essence of which is exceedingly short feedback loops and the ability to pivot as needs change?

This paper conveys the experience of Lockheed Martin, the world's largest defense contractor, as it is applying PLE and Agile together on one of its largest and most important projects. Not only is the project highly visible with demanding requirements, it is also very large, comprising some 10 million lines of code. This setting would challenge either methodology by itself; putting both of them together is yielding many lessons. At the end of

Integration achieved by applying agile practices within small teams assigned to deliver each product.

Our Goal was different…

We started with an agile process that allowed us to incrementally add features…..

and we wanted to continue with an agile process in order to progressively add variability points..

Agile: XP Style

1. On-site customer
2. User stories
3. Short cycles
4. Acceptance tests
5. Pairs programming
6. Test-driven
7. Collective ownership
8. Continuous integration
9. Sustainable pace
10. Open workspace
11. The planning game
12. Simple design
13. Refactoring
14. Metaphor

EARS: Easy Requirements Specifications

<optional preconditions> <optional trigger>
the <system name> shall
<system response>

1. **Ubiquitous**
   Requirement is always active
2. **Event-driven** (keyword When)
   Required response to a triggering event
3. **State-driven** (keyword While)
   Required response in a specified state
4. **Option** (keyword Where)
   Applicable only if feature is included
5. **Hybrid**: Use combinations of when, while and where for requirements with complex conditional clauses.

- Simple structure adds rigor & clarity
- System response describes what the system must actually do that is *visible* at the boundary of the system

Alistair Mavin, Philip Wilkinson, Adrian Harwood, Mark Novak: Easy Approach to Requirements Syntax (EARS). IEEE International Conference on Requirements Engineering. 2009: 317-322

EARS “Stories”

Ubiquitous

The *<component name>* shall *<response>*

The drone shall maintain a *minimum-separation distance* at all times.

Option

Where <feature is included> the <system name> shall <system response>

Where parachute mode is enabled and a drop is initiated the drone shall scan the dropzone for obstacles.

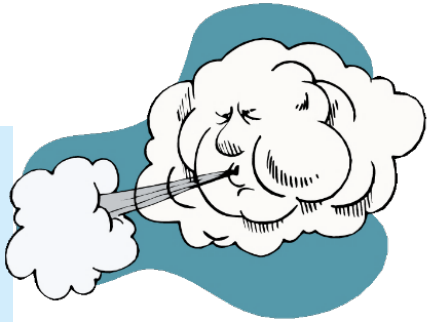
Event Driven.

When *<trigger>* the *<system name>* shall *<system response>*

When the drone is within X centimeters of minimum separation distance from another drone, the collision avoidance system shall provide directives to all drones in the vicinity.

Unwanted Behavior

If <optional preconditions> <trigger>, then the <system name> shall <system response>



If wind gusts exceed desired wind velocity but are below the maximum wind velocity, the drone shall return to base.

State Driven

While *<in a specific state>* the *<system name>* shall *<system response>*

While in landing mode the drone shall descend vertically until it reaches the ground.

Fault Mode Effect Criticality Analysis (FMECA)

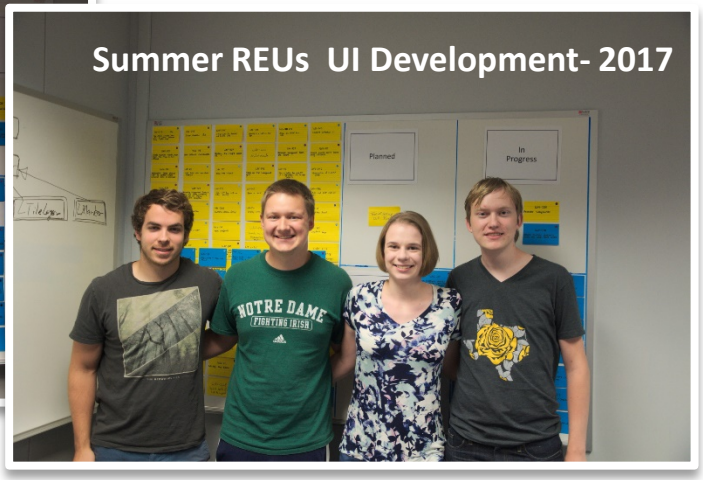
Preliminary Hazard:
Drone's battery fails unexpectedly and the drone crashes.

❶ The battery level **detector** fails to detect a low battery level.

❷ Battery level **indicator** fails and an incorrect battery level is reported.

❸ The software fails to check battery level in time to take responsive actions before the battery fails.

“Safety” Scrum



Inputs from Stakeholders, Customers, Users, Team

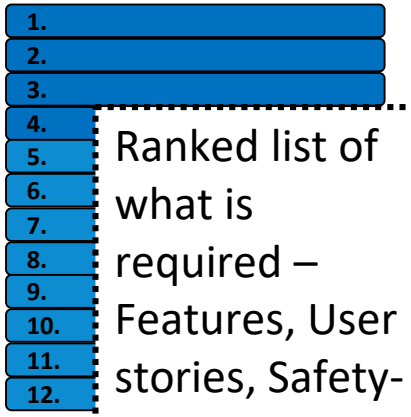
Preliminary Hazard Analysis

Safety Analysts

Preliminary FMECA (Fault Mode Criticality Analysis)

Product Owner

Scrum meeting

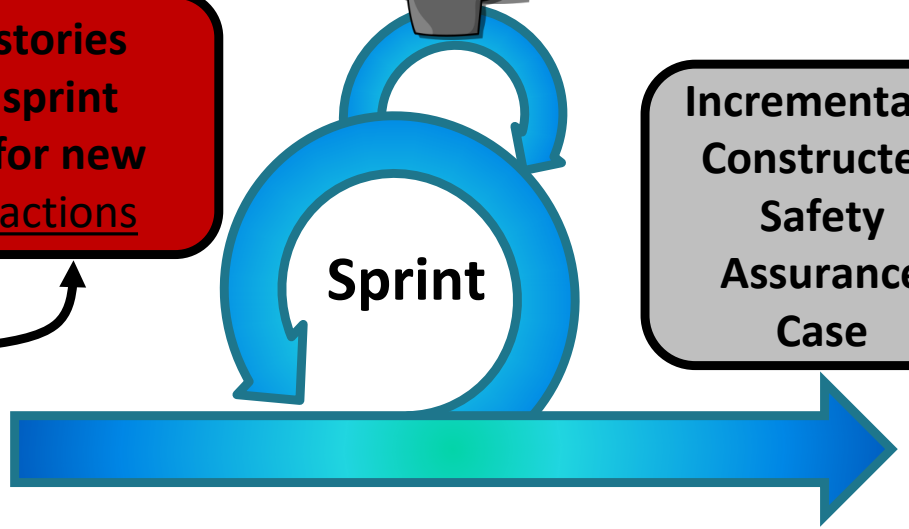


Ranked list of what is required – Features, User stories, Safety-Stories

Product Backlog

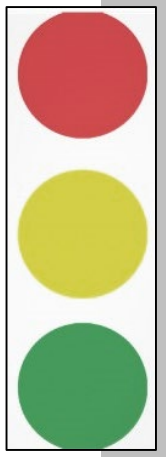
FMECA for stories placed into sprint backlog and for new feature interactions

Team selects user stories for the sprint



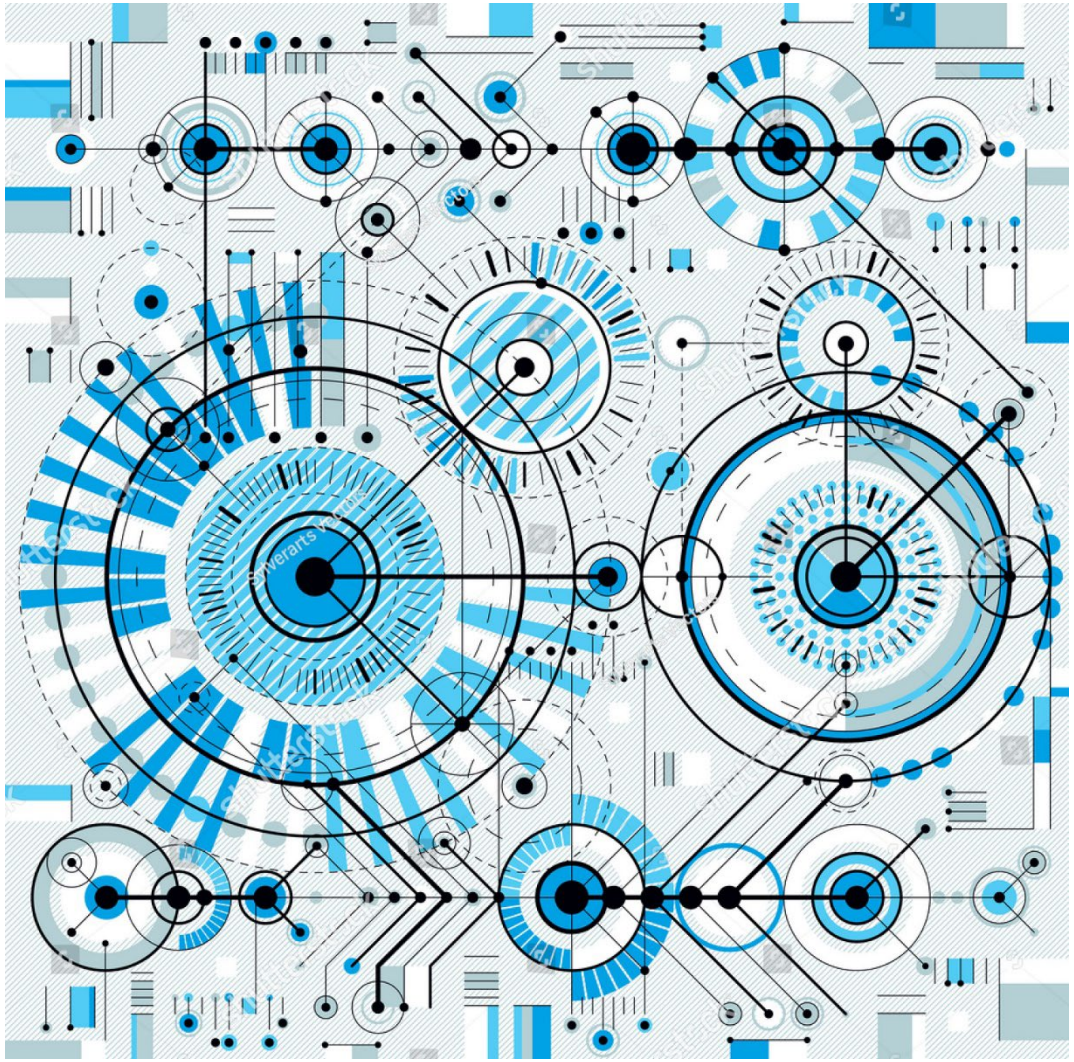
Incrementally Constructed Safety Assurance Case

Is it current ly safe?



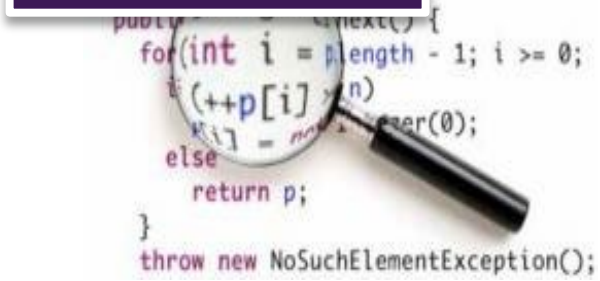
Working Product

Reverse Engineering



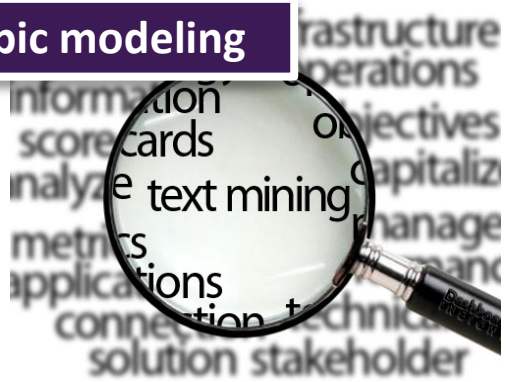
What does the literature say?

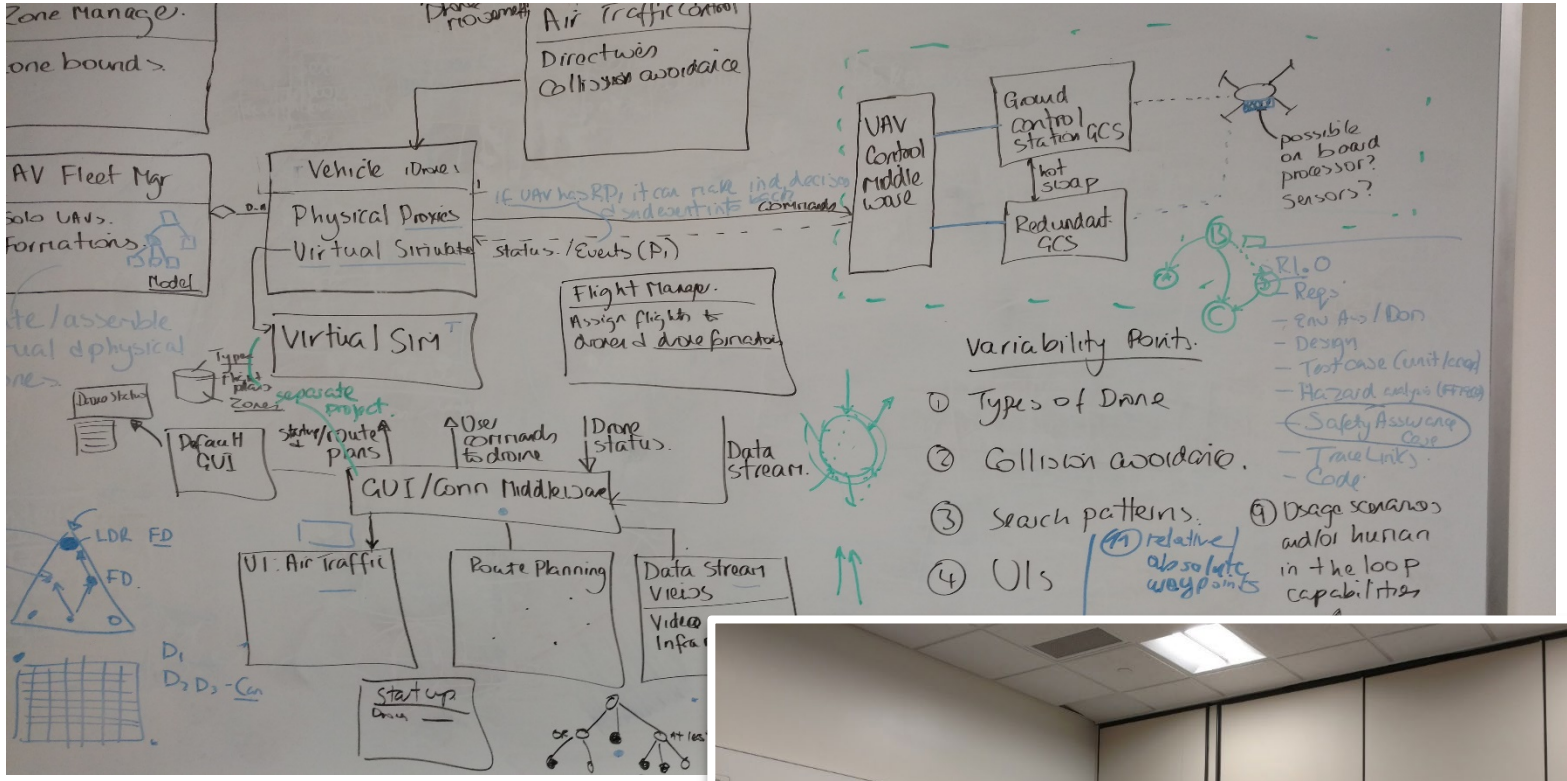
Static analysis



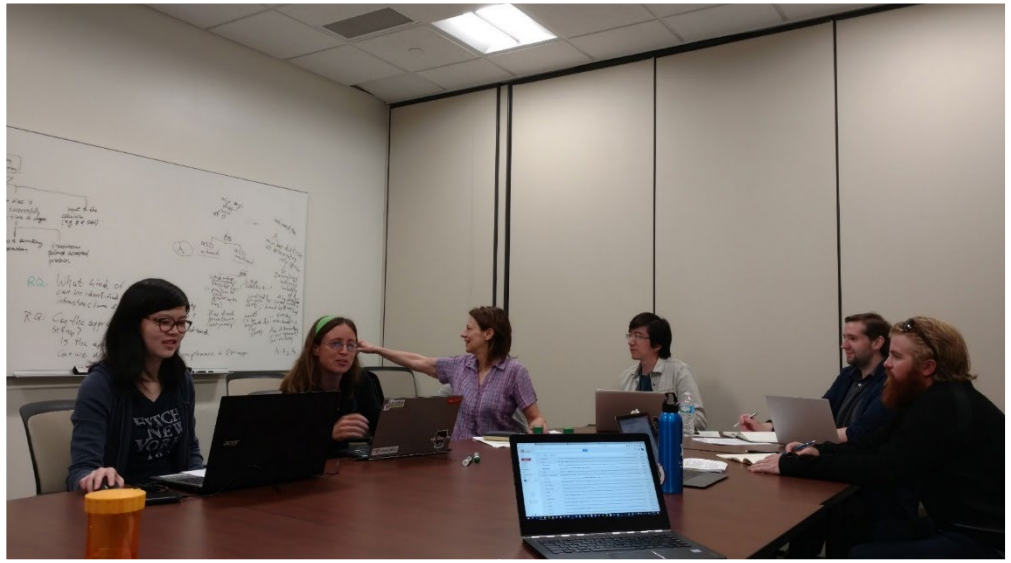
Runtime trace

Topic modeling



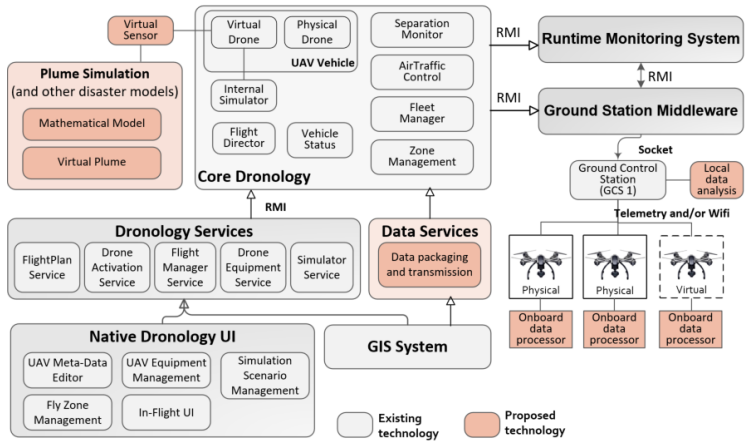


But fail to fully leverage human knowledge of the ongoing project.



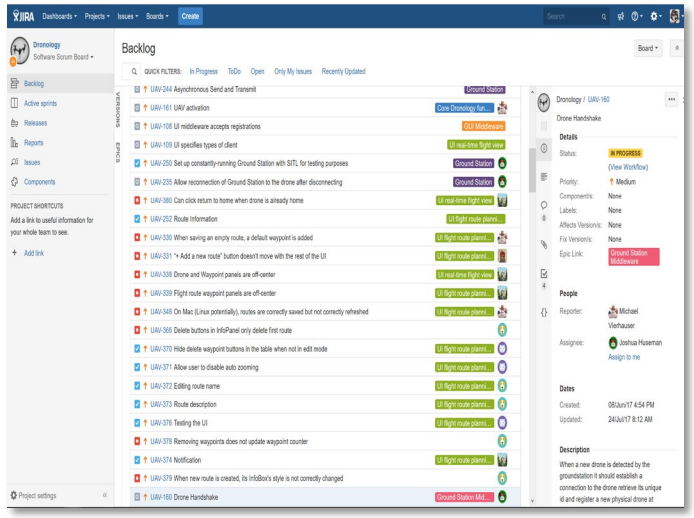
Bottom up discovery

Bottom up



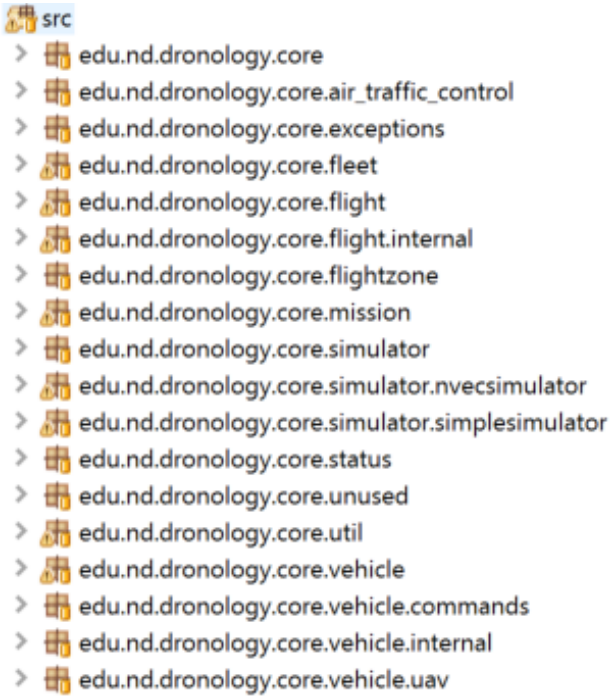
❶ **Architecture:** Intuitive, visual, component-based perspective. *To what extent do **components** map to features & variability points?*

Bottom up



❷ **Requirements:** Mapped To epics. *To what extent do **epics** map to features and their variability points?*

Bottom Up

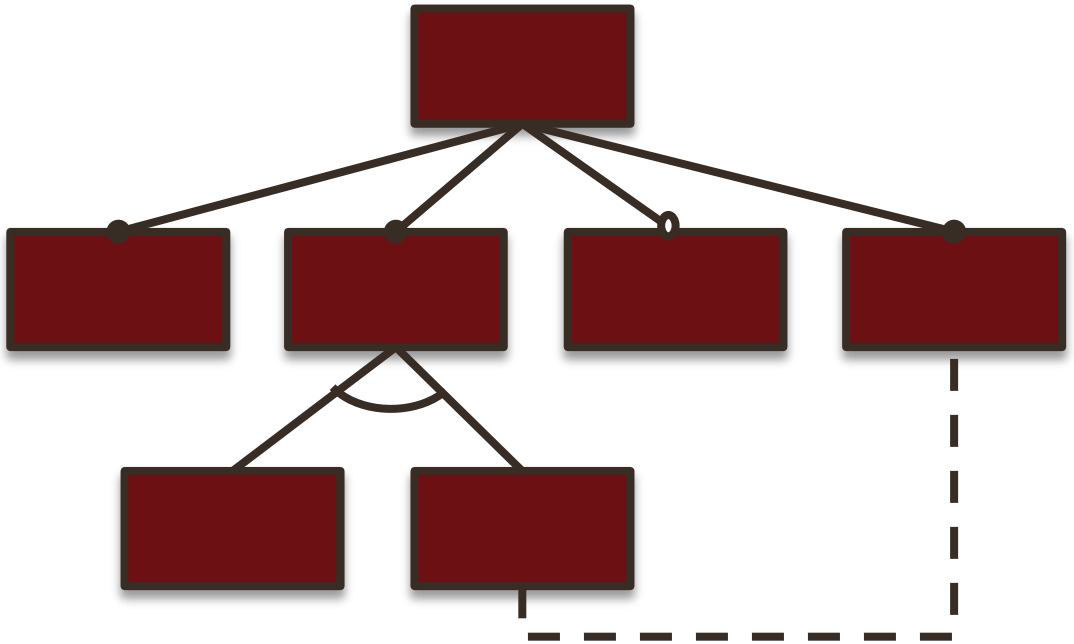


❸ **Source Code:** Identify packages. Do they map to features?

All of these have “clues” about possible features…

Top Down

Top down



❹ **Features & Variability Points:**
What are the variability points?

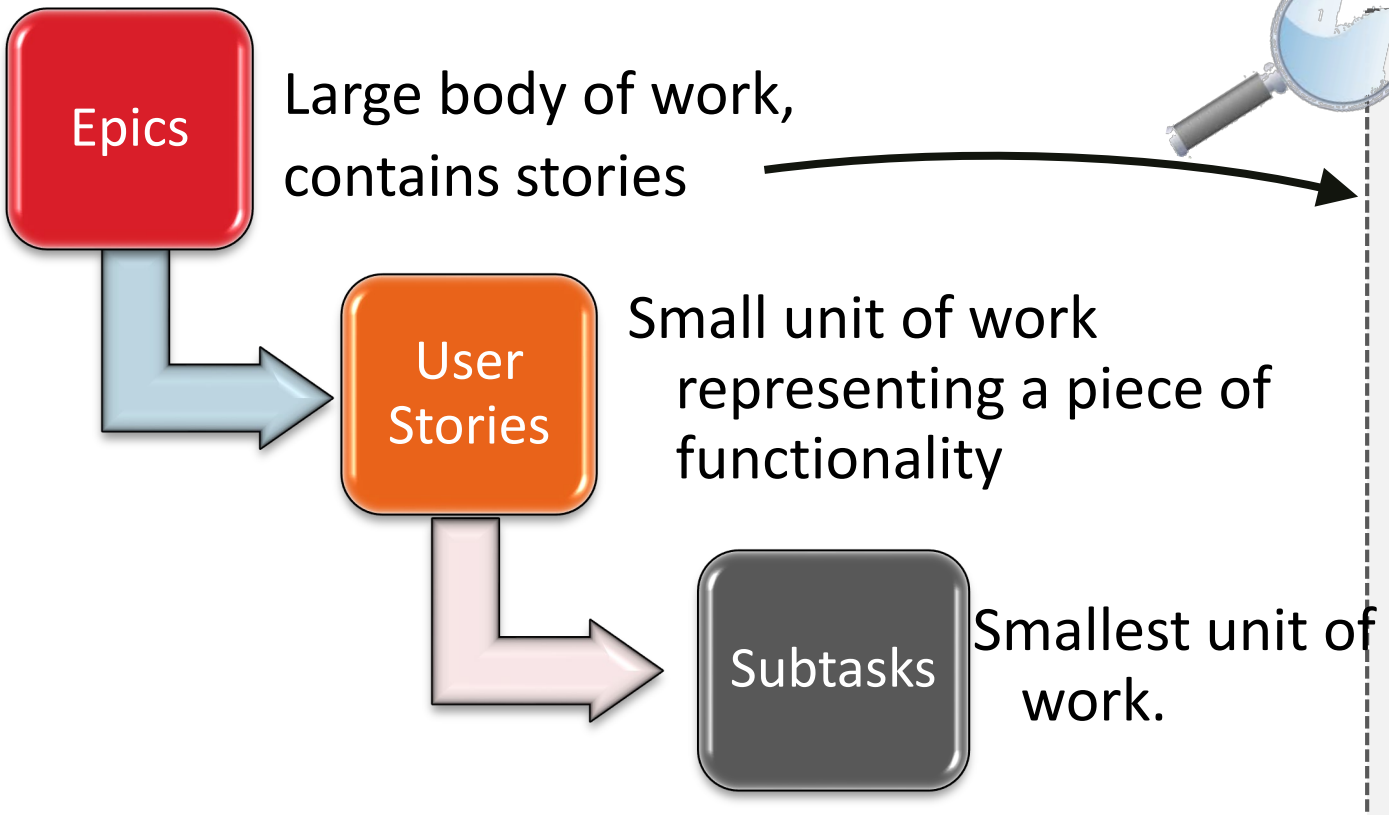
Requirements Engineeri
researchers interested in
Dronology to support res
into traceability, forensic
requirements, goal mode
runtime adaptation....

End users interested in deploying Dronology to support specific scenarios of use.

Elicit stakeholder's needs. Perform domain analysis.

A fully top-down approach could imply a break-away from agile.

Clues for Commonalities and Variabilities in Jira



- Simulator
- UI flight route planning view
- Safety Related Issues
- Collision Avoidance
- UI real-time flight view
- Supporting artifacts
- Ground Station
- Ground Station Middleware
- UAV Hardware
- Flight route management
- Core Dronology - UAV management
- Manage core dronology functionality
- Manage interface between Dronology and GUI clients
- Platform, Language, and Portability issues
- Vaadin

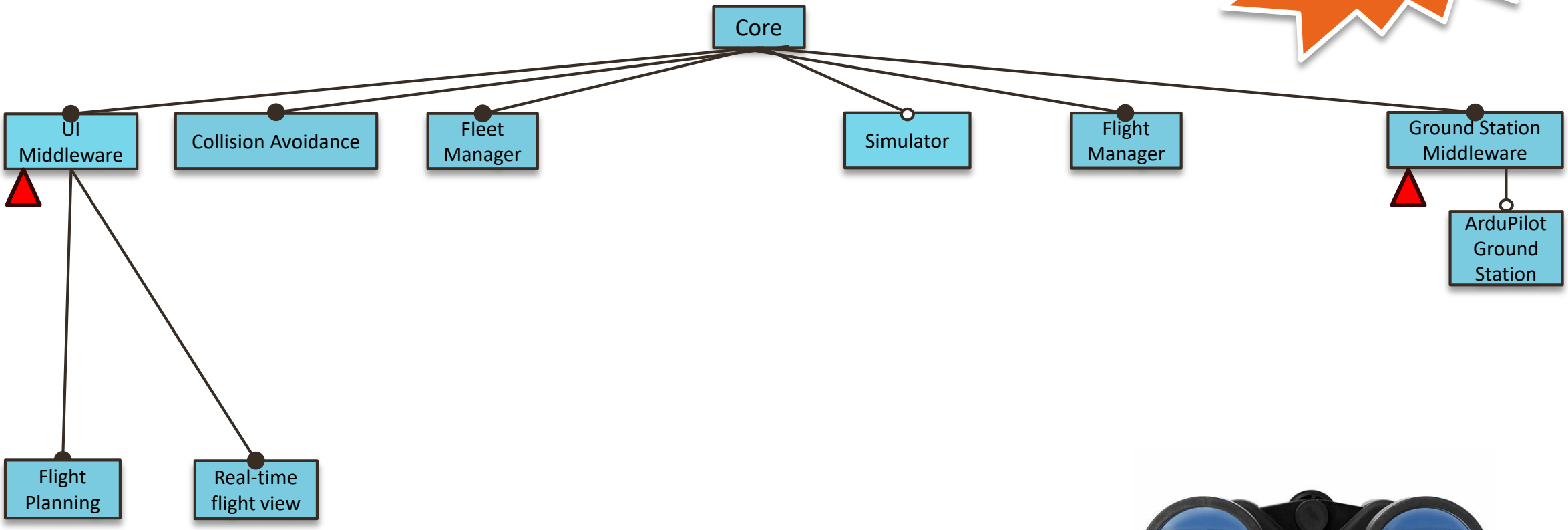
Places that we deliberately built for variability

Epics with obvious potential for variability

Epics representing commonalities

As-Is Product (Jira)

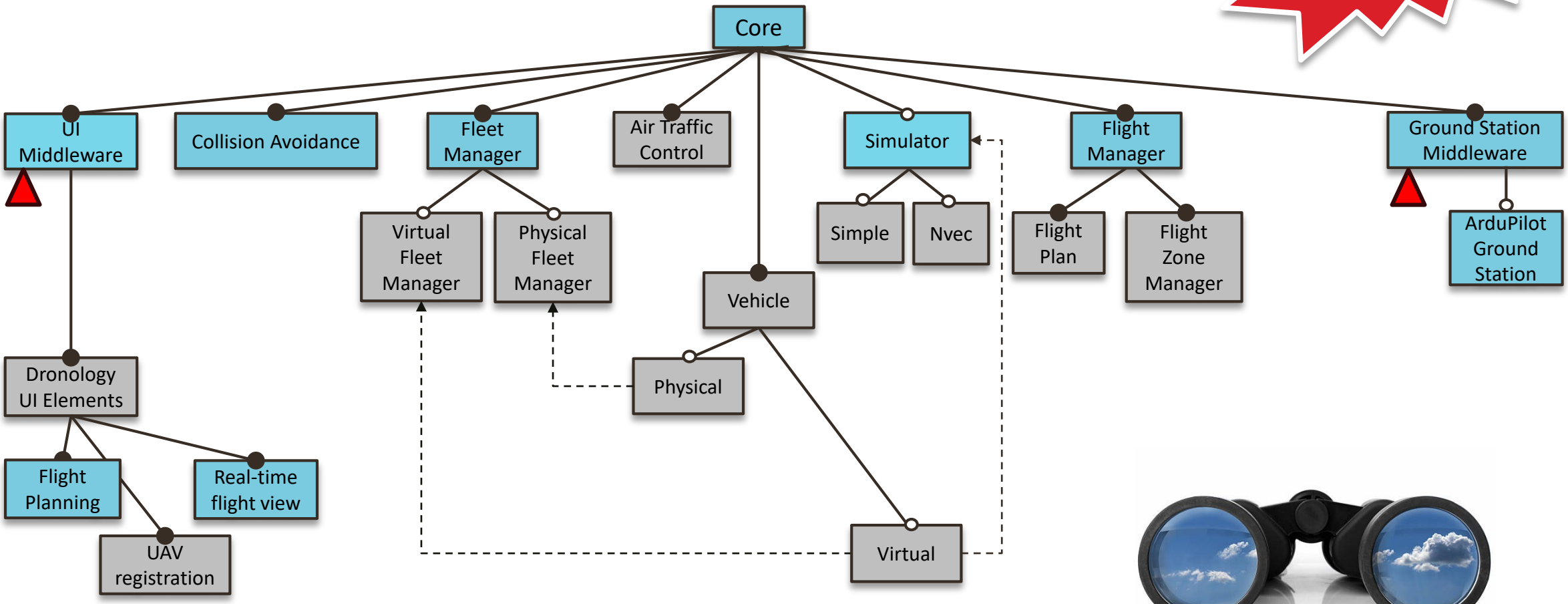
As is product!



The world as Jira sees it!

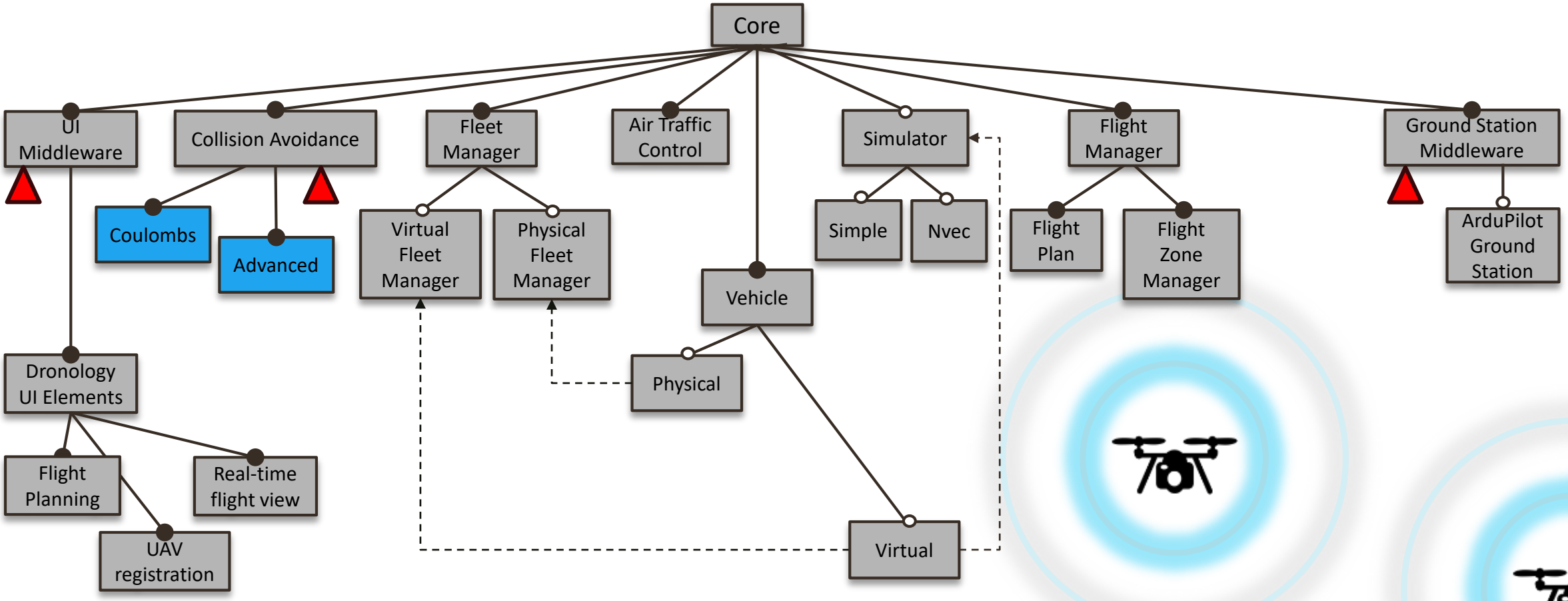
As-is Product (Jira + Packages)

As is product!



Augmented by a **code-level package view.**

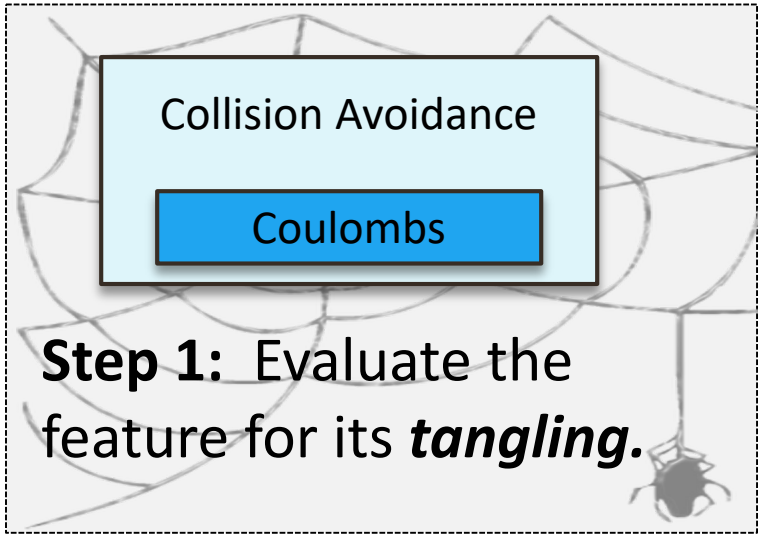
Refining the Core Feature Model



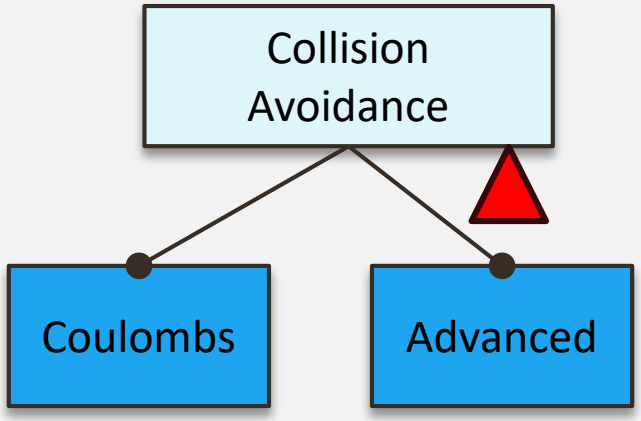
Adding a new feature is really an extensive form of impact and **untangling** analysis.

Treat it as a user story with sub-tasks or as an epic.

When variability is not envisioned..



Step 1: Evaluate the feature for its ***tangling.***



Step 2: Envision the future.

Step 3: Write untangling stories.

Put the user story and subtask into the backlog..

❶ As a *researcher* I want to exchange collision avoidance algorithms. (when?)

❷ When CollisionAvoidance_HotSwap event occurs, the collision avoidance strategy shall be exchanged at runtime.

Tasks:

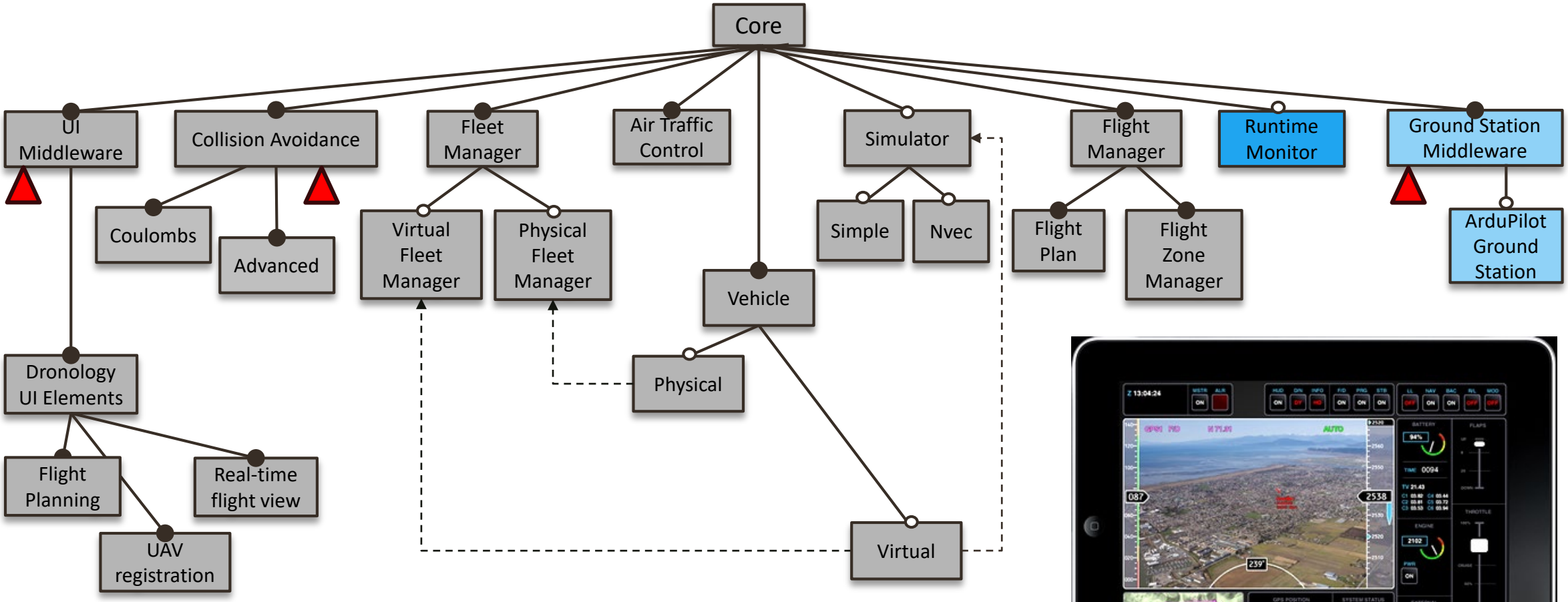
- Define collision avoidance interface
- Extract concrete collision avoidance implementation to a concrete strategy class.
- Push code shared across collision avoidance algorithms to a common class.

Safety Critical Product Line

What new safety requirements are introduced as a result of the new variability point?

Perform safety hazard analysis on hot swapping collision avoidance algorithms.

Adding a new Feature

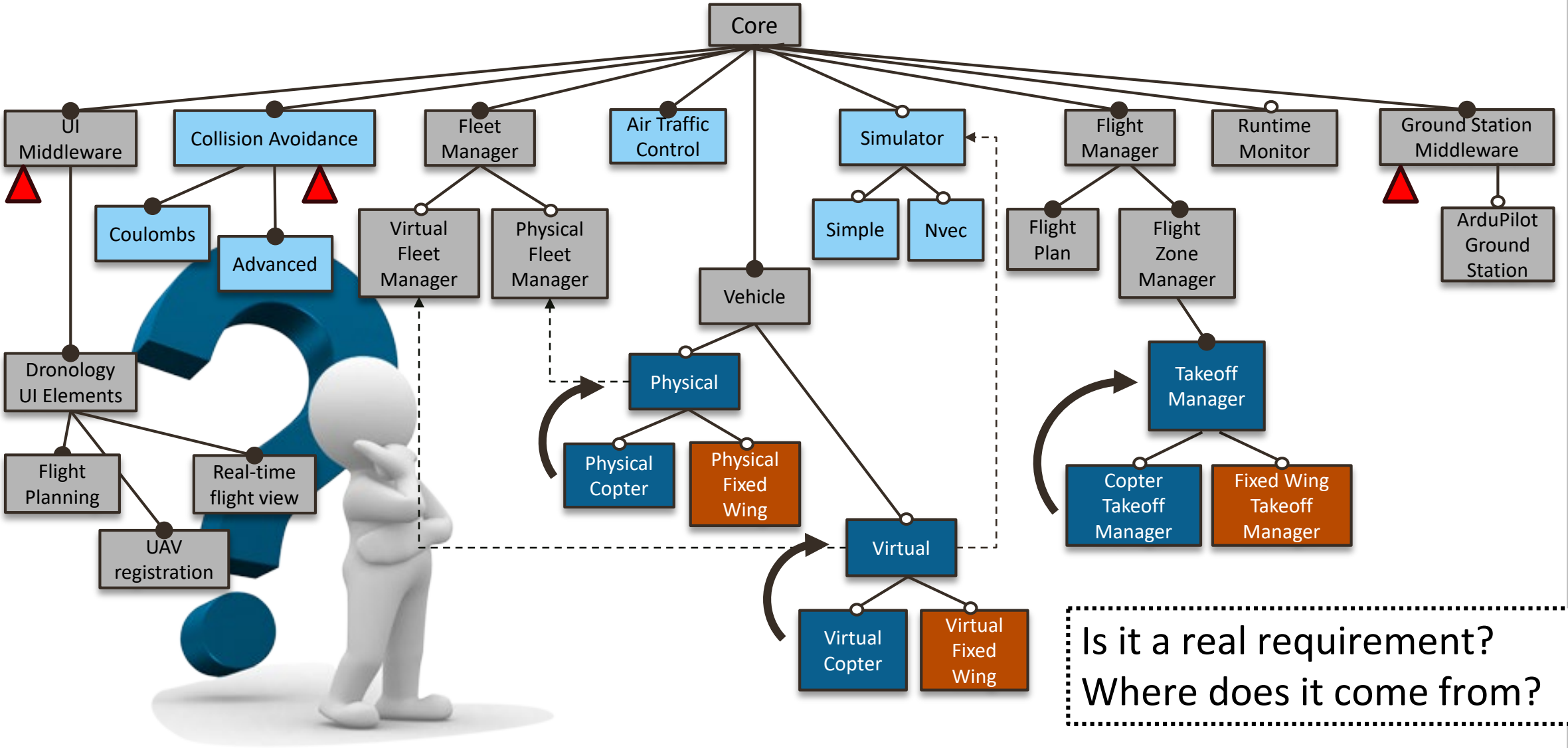


Architecturally Significant Variability Points

Architecturally significant requirements are :

- Technically challenging or constraining.
- Have a pervasive impact on the system.
- Often, but not always, related to -ilities

Adding an architecturally significant variability point

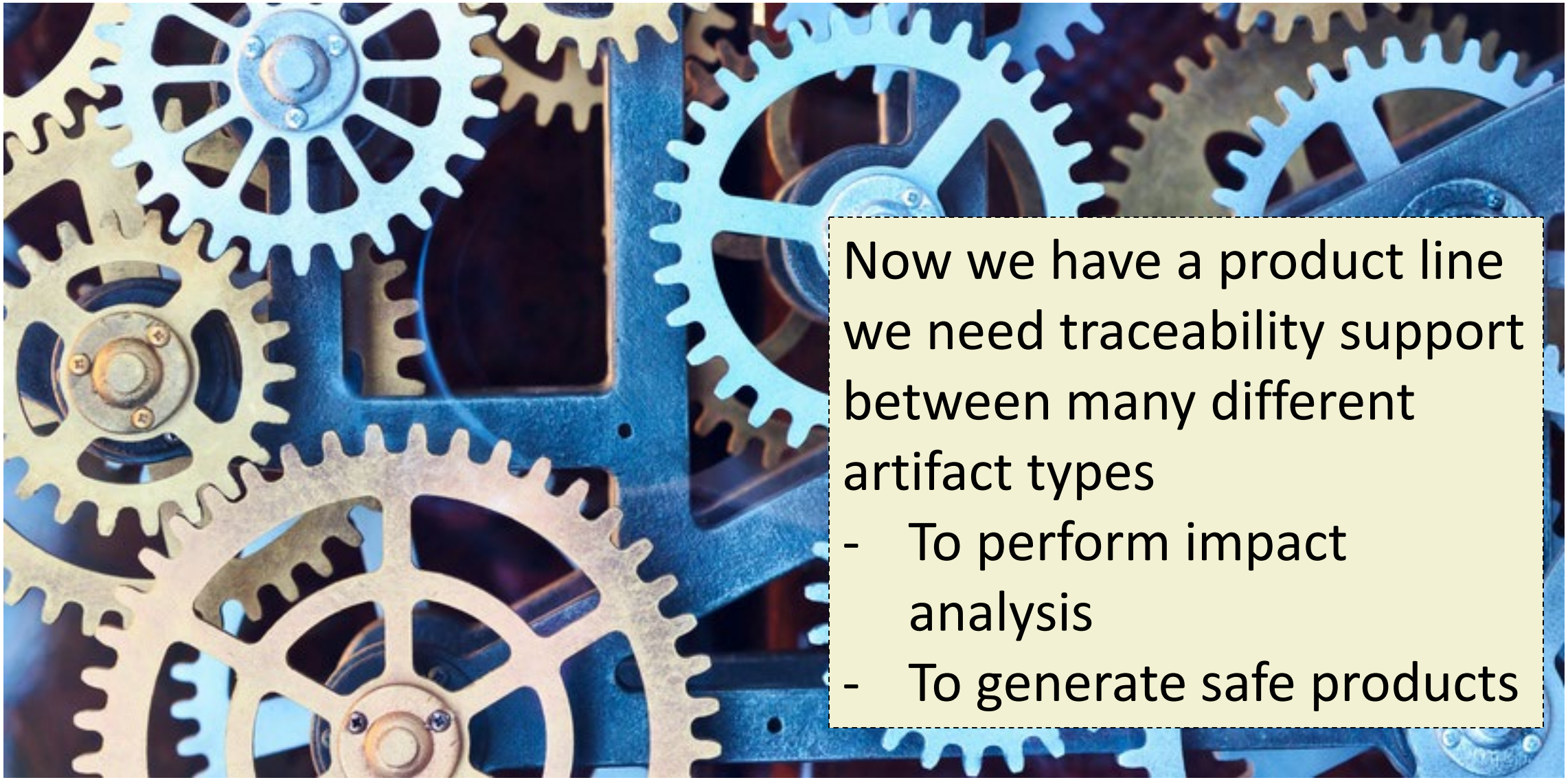


Is it a real requirement?
Where does it come from?

Is it a real requirement?

When do we invest in it?

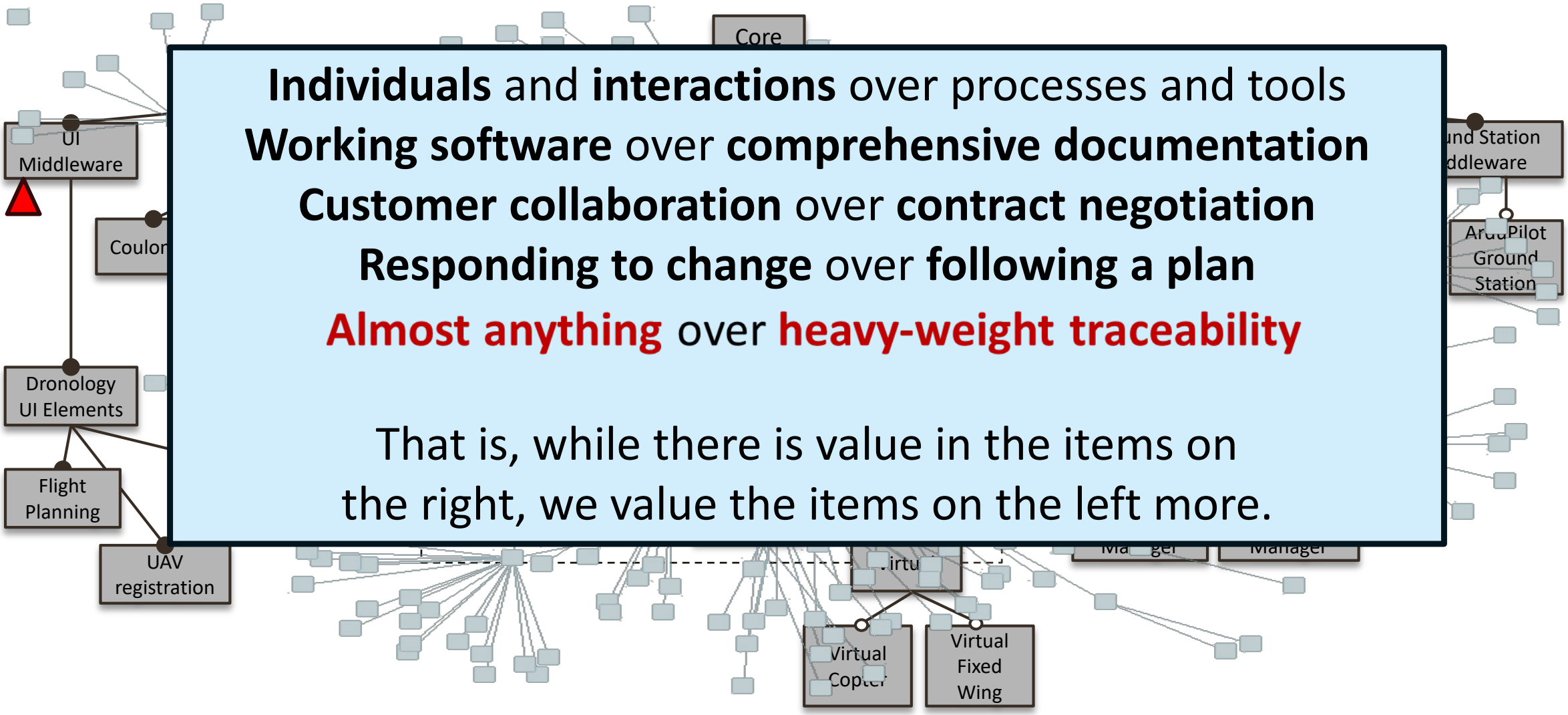
Next steps



Now we have a product line we need traceability support between many different artifact types

- To perform impact analysis
- To generate safe products

How do we configure and maintain an Agile PL?



Individuals and **interactions** over processes and tools

Working software over **comprehensive documentation**

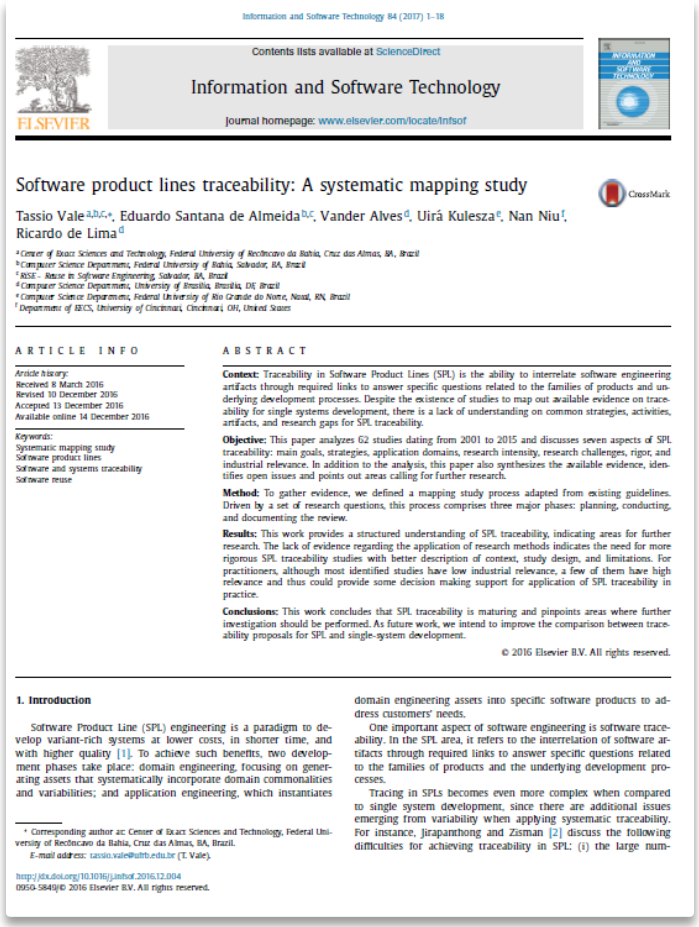
Customer collaboration over **contract negotiation**

Responding to change over **following a plan**

Almost anything over **heavy-weight traceability**

That is, while there is value in the items on the right, we value the items on the left more.

Traceability in Agile SPLs



Information and Software Technology 84 (2017) 1–18

Contents lists available at ScienceDirect

Information and Software Technology

Journal homepage: www.elsevier.com/locate/infsof

Software product lines traceability: A systematic mapping study

Tassio Vale[a,b,c,*], Eduardo Santana de Almeida[b,c], Vander Alves[d], Uirá Kulesza[e], Nan Niu[f], Ricardo de Lima[d]

[a] Center of Exact Sciences and Technology, Federal University of Recôncavo da Bahia, Cruz das Almas, BA, Brazil
[b] Computer Science Department, Federal University of Bahia, Salvador, BA, Brazil
[c] RISE - Reuse in Software Engineering, Salvador, BA, Brazil
[d] Computer Science Department, University of Brasília, Brasília, DF, Brazil
[e] Computer Science Department, Federal University of Rio Grande do Norte, Natal, RN, Brazil
[f] Department of EECS, University of Cincinnati, Cincinnati, OH, United States

ARTICLE INFO

Article history:
Received 8 March 2016
Revised 10 December 2016
Accepted 13 December 2016
Available online 14 December 2016

Keywords:
Systematic mapping study
Software product lines
Software and systems traceability
Software reuse

ABSTRACT

Context: Traceability in Software Product Lines (SPL) is the ability to interrelate software engineering artifacts through required links to answer specific questions related to the families of products and underlying development processes. Despite the existence of studies to map out available evidence on traceability for single systems development, there is a lack of understanding on common strategies, activities, artifacts, and research gaps for SPL traceability.

Objective: This paper analyzes 62 studies dating from 2001 to 2015 and discusses seven aspects of SPL traceability: main goals, strategies, application domains, research intensity, research challenges, rigor, and industrial relevance. In addition to the analysis, this paper also synthesizes the available evidence, identifies open issues and points out areas calling for further research.

Method: To gather evidence, we defined a mapping study process adapted from existing guidelines. Driven by a set of research questions, this process comprises three major phases: planning, conducting, and documenting the review.

Results: This work provides a structured understanding of SPL traceability, indicating areas for further research. The lack of evidence regarding the application of research methods indicates the need for more rigorous SPL traceability studies with better description of context, study design, and limitations. For practitioners, although most identified studies have low industrial relevance, a few of them have high relevance and thus could provide some decision making support for application of SPL traceability in practice.

Conclusions: This work concludes that SPL traceability is maturing and pinpoints areas where further investigation should be performed. As future work, we intend to improve the comparison between traceability proposals for SPL and single-system development.

1. Introduction

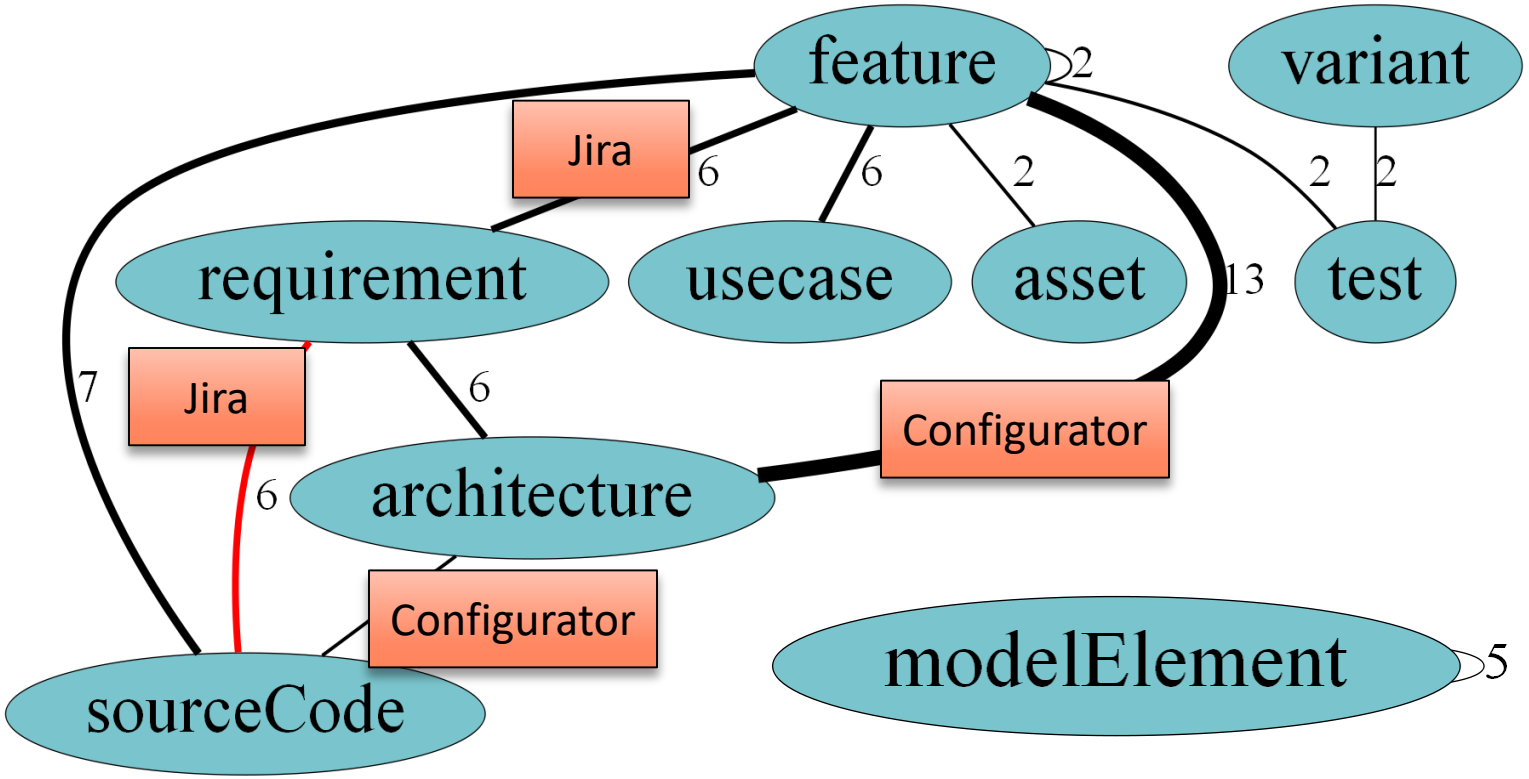
Software Product Line (SPL) engineering is a paradigm to develop variant-rich systems at lower costs, in shorter time, and with higher quality [1]. To achieve such benefits, two development phases take place: domain engineering, focusing on generating assets that systematically incorporate domain commonalities and variabilities; and application engineering, which instantiates domain engineering assets into specific software products to address customers' needs.

One important aspect of software engineering is software traceability. In the SPL area, it refers to the interrelation of software artifacts through required links to answer specific questions related to the families of products and the underlying development processes.

Tracing in SPLs becomes even more complex when compared to single system development, since there are additional issues emerging from variability when applying systematic traceability. For instance, Jirapanthong and Zisman [2] discuss the following difficulties for achieving traceability in SPL: (i) the large num-

* Corresponding author at: Center of Exact Sciences and Technology, Federal University of Recôncavo da Bahia, Cruz das Almas, BA, Brazil.
E-mail address: tassio.vale@ufrb.edu.br (T. Vale).

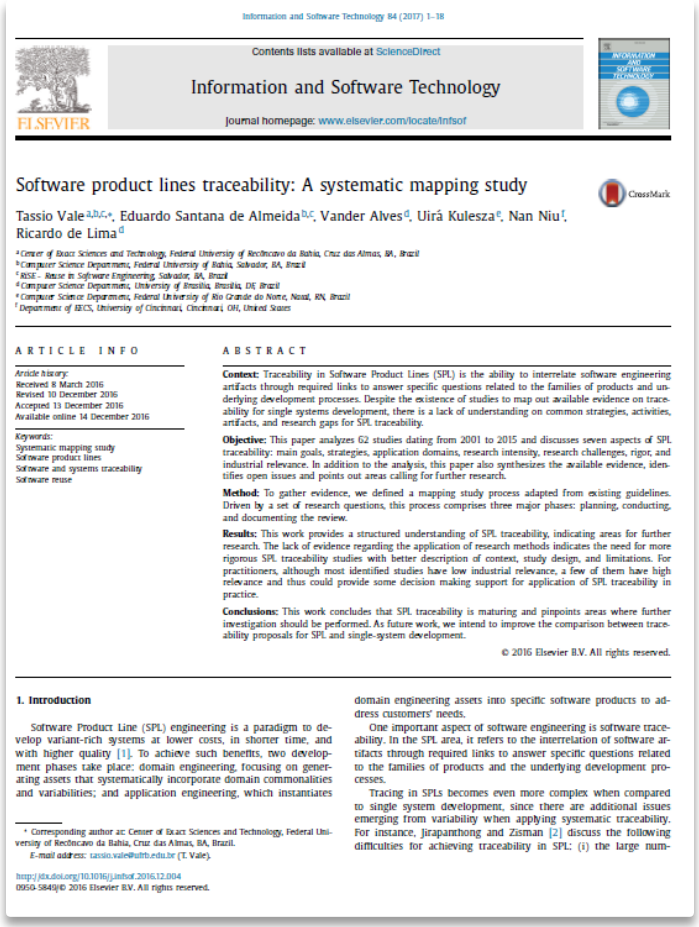
http://dx.doi.org/10.1016/j.infsof.2016.12.004
0950-5849/© 2016 Elsevier B.V. All rights reserved.



Based on an analysis of 53 papers which referenced product-line traceability. Only showing links that were mentioned more than one time.

Goal: Leverage agile tracing practices without adding additional overhead.

Traceability in Agile SPLs



Information and Software Technology 84 (2017) 1–18

Contents lists available at ScienceDirect

Information and Software Technology

Journal homepage: www.elsevier.com/locate/infsof

Software product lines traceability: A systematic mapping study

Tassio Vale[a,b,c,*], Eduardo Santana de Almeida[b,c], Vander Alves[d], Uirá Kulesza[e], Nan Niu[f], Ricardo de Lima[d]

[a] Center of Exact Sciences and Technology, Federal University of Recôncavo da Bahia, Cruz das Almas, BA, Brazil
[b] Computer Science Department, Federal University of Bahia, Salvador, BA, Brazil
[c] RiSE - Reuse in Software Engineering, Salvador, BA, Brazil
[d] Computer Science Department, University of Brasília, Brasília, DF, Brazil
[e] Computer Science Department, Federal University of Rio Grande do Norte, Natal, RN, Brazil
[f] Department of EECS, University of Cincinnati, Cincinnati, OH, United States

ARTICLE INFO

Article history:
Received 8 March 2016
Revised 10 December 2016
Accepted 13 December 2016
Available online 14 December 2016

Keywords:
Systematic mapping study
Software product lines
Software and systems traceability
Software reuse

ABSTRACT

Context: Traceability in Software Product Lines (SPL) is the ability to interrelate software engineering artifacts through required links to answer specific questions related to the families of products and underlying development processes. Despite the existence of studies to map out available evidence on traceability for single systems development, there is a lack of understanding on common strategies, activities, artifacts, and research gaps for SPL traceability.

Objective: This paper analyzes 62 studies dating from 2001 to 2015 and discusses seven aspects of SPL traceability: main goals, strategies, application domains, research intensity, research challenges, rigor, and industrial relevance. In addition to the analysis, this paper also synthesizes the available evidence, identifies open issues and points out areas calling for further research.

Method: To gather evidence, we defined a mapping study process adapted from existing guidelines. Driven by a set of research questions, this process comprises three major phases: planning, conducting, and documenting the review.

Results: This work provides a structured understanding of SPL traceability, indicating areas for further research. The lack of evidence regarding the application of research methods indicates the need for more rigorous SPL traceability studies with better description of context, study design, and limitations. For practitioners, although most identified studies have low industrial relevance, a few of them have high relevance and thus could provide some decision making support for application of SPL traceability in practice.

Conclusions: This work concludes that SPL traceability is maturing and pinpoints areas where further investigation should be performed. As future work, we intend to improve the comparison between traceability proposals for SPL and single-system development.

1. Introduction

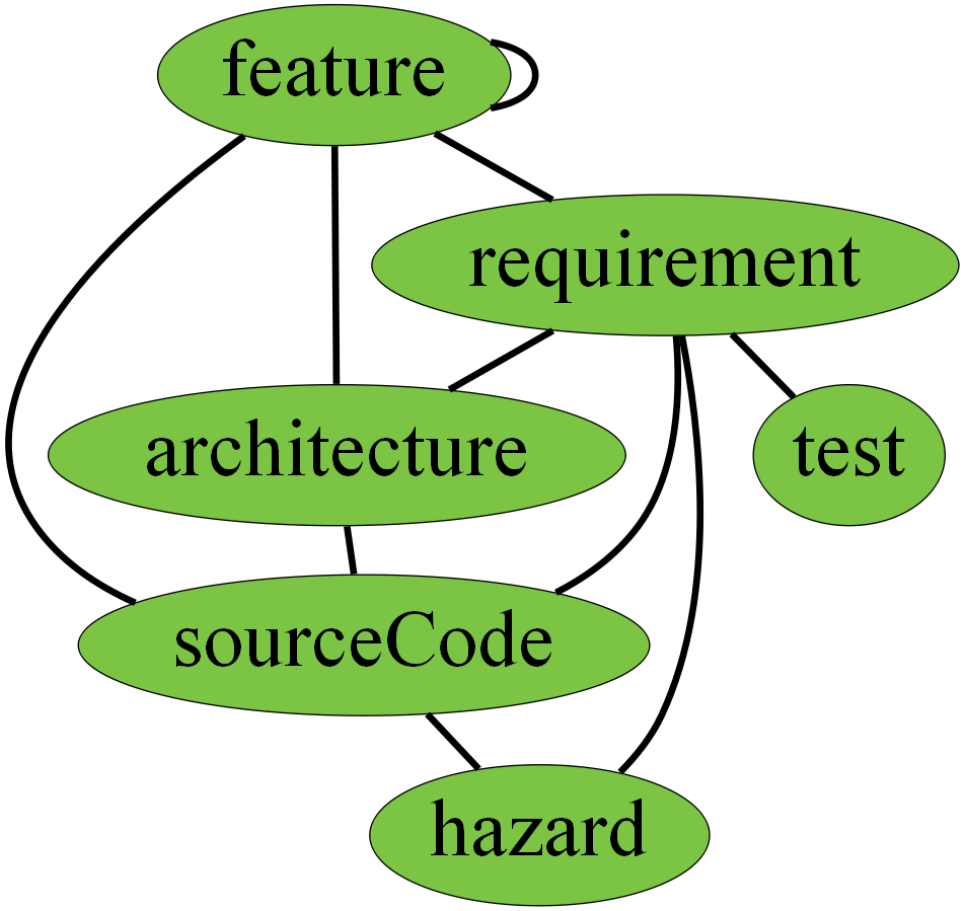
Software Product Line (SPL) engineering is a paradigm to develop variant-rich systems at lower costs, in shorter time, and with higher quality [1]. To achieve such benefits, two development phases take place: domain engineering, focusing on generating assets that systematically incorporate domain commonalities and variabilities; and application engineering, which instantiates domain engineering assets into specific software products to address customers' needs.

One important aspect of software engineering is software traceability. In the SPL area, it refers to the interrelation of software artifacts through required links to answer specific questions related to the families of products and the underlying development processes.

Tracing in SPLs becomes even more complex when compared to single system development, since there are additional issues emerging from variability when applying systematic traceability. For instance, Jirapanthong and Zisman [2] discuss the following difficulties for achieving traceability in SPL: (i) the large num-

* Corresponding author at: Center of Exact Sciences and Technology, Federal University of Recôncavo da Bahia, Cruz das Almas, BA, Brazil.
E-mail address: tassio.vale@ufrb.edu.br (T. Vale).

http://dx.doi.org/10.1016/j.infsof.2016.12.004
0950-5849/I© 2016 Elsevier B.V. All rights reserved.

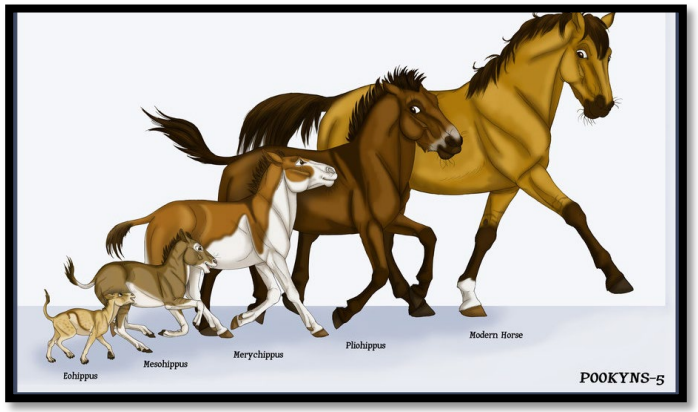


Goal: Leverage agile tracing practices without adding additional overhead.

How do we achieve this in an agile way?

Tag commits – and infer trace links from them.

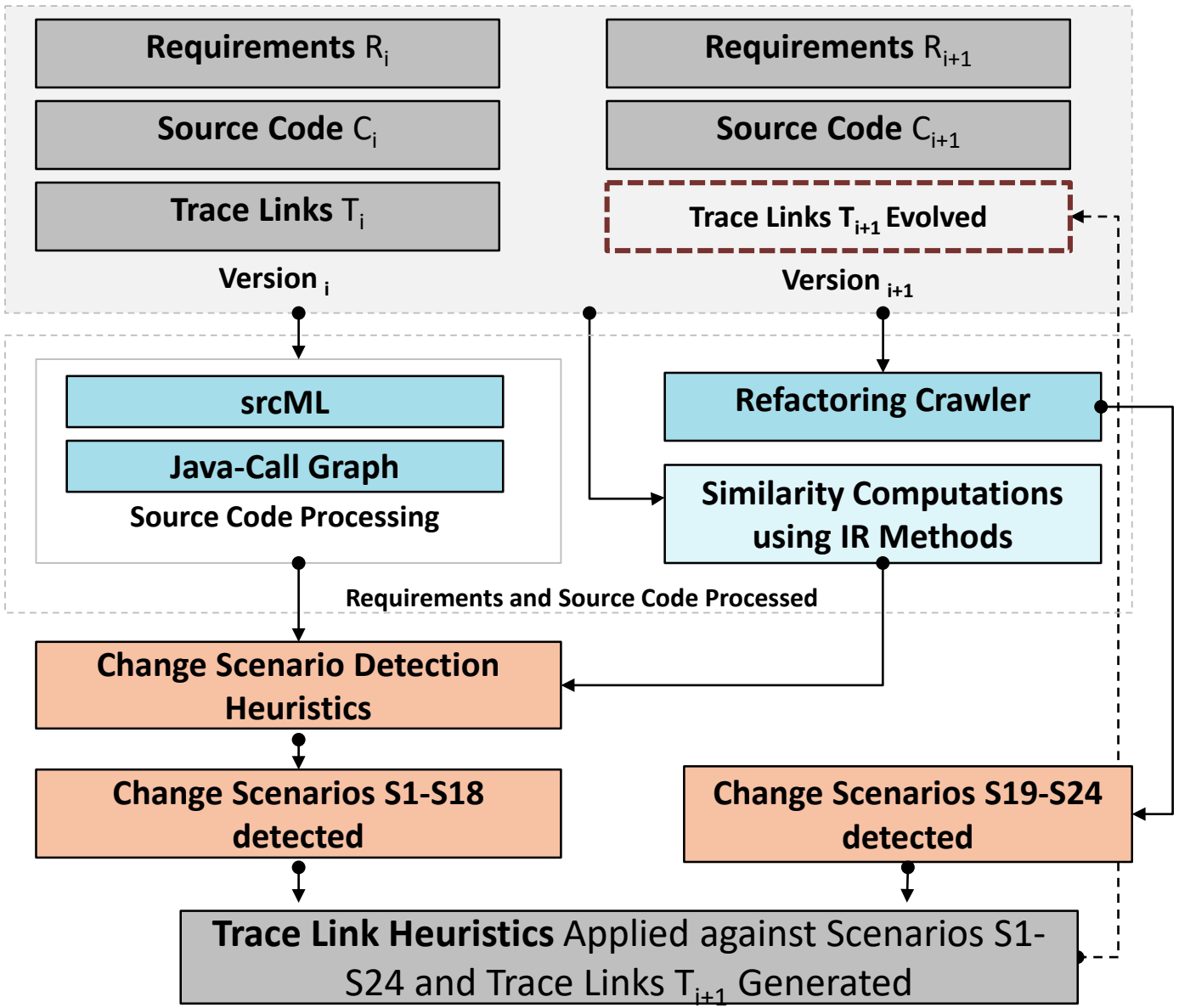
Fill in the gaps by recommending miss-ing tags.



Evolve trace links automatically.

Try to build traceability into the daily process to the fullest extent possible

TLE: How it works



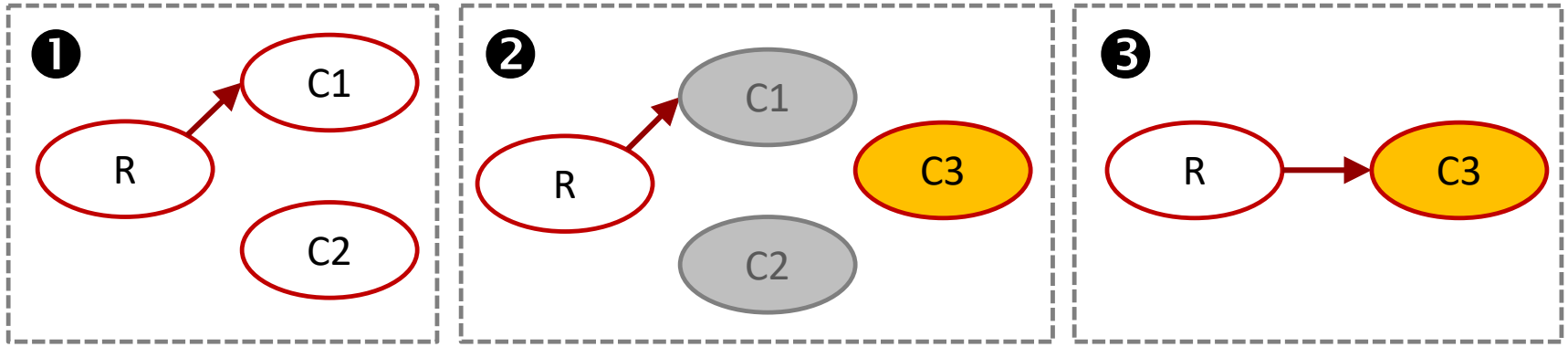
❶ Software artifacts changed across versions

❷Tools for detecting changes in code and requirements.

❸ TLE tools and algorithms for recognizing change patterns and evolving trace links.

Identify Patterns of change

P_1: Added Class New Functionality	P_2: Added Method New Functionality	P_3: Modified Method New Functionality	P_7: Added Class Extracted Class	P_8: Added Class Merged Classes	P_9: Added Class Promoted Method	P_{13}: Deleted Class Merged Classes	P_{14}: Added Method Extracted Method	P_{15}: Added Method Merged Methods
Pattern Description: New functionality appears in form of a new class.	**Pattern Description:** New functionality appears in form of a new method.	**Pattern Description:** New functionality appears in form of a modified method.	**Pattern Description:** The new class is extracted from an existing one.	**Pattern Description:** The new class represents a merging of existing ones.	**Pattern Description:** The new class is created as a result of promoting method.	**Pattern Description:** The deleted classes are merged into a new class.	**Pattern Description:** The new method is an old promoted method.	**Pattern Description:** The new method represents a merging of old ones.
P_4: Deleted Class Obsolete Functionality	P_5: Deleted Method Obsolete Functionality	P_6: Modified Method Obsolete Functionality	P_{10}: Added Class Extracted Subclass	P_{11}: Added Class Extracted Superclass	P_{12}: Deleted Class Divided Class	P_{16}: Added Method Divided Method	P_{17}: Deleted Method Divided Methods	P_{18}: Deleted Method Merged Methods
Pattern Description: Obsolete functionality is getting deleted in form of a class.	**Pattern Description:** Obsolete functionality is getting deleted in form of a method.	**Pattern Description:** Obsolete functionality is getting deleted in form of a modified method.	**Pattern Description:** New class is an extracted subclass.	**Pattern Description:** New class is an extracted superclass.	**Pattern Description:** The deleted class is divided into new classes.	**Pattern Description:** New method is a division of an old method.	**Pattern Description:** The deleted method is divided into new methods.	**Pattern Description:** The deleted methods are merged into a new method.



Two classes are merged. The original link is reassigned to the merged class.

Mona Rahimi, Jane Cleland-Huang: Ptterns of co-evolution between requirements and source code. RePa@RE 2015: 25-31

Detecting a Scenario

Rule Type		Artifact Properties	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}
Add Class Delete Class Add Method Del Method Mod Method	1	$\exists C_{i+1} \mid \nexists C_i$	•	•	•	•	•	•												
	2	$\nexists C_{i+1} \mid \exists C_i$							•	•	•									
	3	$\exists m_{i+1} \mid \nexists m_i$										•	•	•	•					
	4	$\nexists m_{i+1} \mid \exists m_i$														•	•	•		
	5	$\exists m_i \wedge \exists m_{i+1} \mid \nexists Sim(m_i, m_{i+1})$																	•	•
Checks for links between classes	6	$\exists Sim(C_{i+1}, C'_i)$	•			•	•	•					•		•					
	7	$\exists Sim(C_{i+1}, C'_i) \wedge \exists Sim(C_{i+1}, C''_i)$			•									•						
	8																•			
	9																	•		
Checks for links between classes and requirements	10	$\exists Sim(C_{i+1}, R_{i+1})$	•									•							•	
	11	$\forall R \mid Sim(r, C_{i+1}) \subset R_{C'_i}$		•																
	12	$\exists R \mid Sim(r, C_{i+1}) \subseteq R_{m'_i}$											•							
	13	$\forall R \mid Sim(r, C_{i+1}) \subseteq R_{m'_i}$													•					
	14	$\exists R \mid Sim(r, C_{i+1}) \wedge \nexists Sim(r, C'_{i+1})$				•	•	•												
	15	$\forall R \mid Sim(r, C_{i+1}) \subseteq R_{C'} \cup R_{C''}$			•															
	16													•						
	17																			
	18																•			
	19	$R_{C_i} \cup R_{C'_i} \subseteq Sim(r, C''_{i+1})$									•									
	20	$R_{m_i} \cup R_{m'_i} \subseteq Sim(r, C''_{i+1})$																•		
	21	$\forall R \mid Sim(r, C_{i+1}) \subseteq R_{C'_i}$			•															
Checks for methods in classes	22	$\forall m \in C_{i+1} \subset m \in C'_i$	•			•	•	•												
	23	$\forall m \in C_{i+1} \subseteq m \in C'_i \wedge C''_i$		•																
	24										•									
	25								•											
Checks for associations between classes	26							•												
	27	$\exists A(M_{i+1}, m_{i+1}) \subset A(M_i, m'_i)$													•					
	28	$\forall A(M_{i+1}, m_{i+1}) \subseteq A(M_i, m'_i)$											•							
	29	$\forall A(D_{i+1}, C_{i+1}) \subset A(D_i, C'_i) \wedge A(D_i, C''_i)$			•															
	30																			
	31																			
	32																•			
	33	$\forall A(D_i, C_i) \wedge A(D_i, C'_i) \subseteq A(D_i, C''_{i+1})$									•									
	34	$\forall A(M_i, m_i) \wedge A(M_i, m'_i) \subseteq A(M_i, m''_{i+1})$																•		
	35	$\exists A(C_{i+1}, C'_{i+1})$				•	•	•												
	36	$\exists A(C'_{i+1}, C''_{i+1})$								•										
	37	$\exists A(m'_{i+1}, m''_{i+1})$															•			
	38	$\exists A(C_i, C'_i)$									•									
	39	$\exists A(m_i, m'_i)$																•		
Checks the exis of class method and require	40																		•	
	41																			•

$\exists Sim(C_{i+1}, C'_i) \wedge \exists Sim(C_{i+1}, C''_i)$

$\forall R \mid Sim(r, C_{i+1}) \subseteq R_{C'_i} \cup R_{C''_i}$

$\forall m \in C_{i+1} \subseteq m \in C'_i \wedge C''_i$

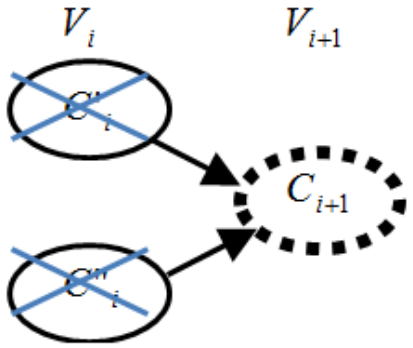
$\forall A(D_{i+1}, C_{i+1}) \subseteq A(D_i, C'_i) \wedge A(D_i, C''_i)$

C_i: a cl
i+1: Sul
R_i: all r
$A(x, y)$:
C_{i+1}: in *Add Class* Scenarios($S_1 - S_6$)is the added class; C_i: in *Delete Class* Scenarios($S_7 - S_9$)is the deleted class
m_{i+1}: in *Add Method* Scenarios($S_{10} - S_{13}$)is the added method; m_i: in *Delete Method* Scenarios($S_{14} - S_{16}$)is to the deleted method

① Use information retrieval techniques, refactoring crawlers, and other static analysis tools to determine which properties hold.

	Link Evolution Actions
S1	$l(C_{i+1}, R_{i+1}) \mid Sim(C_{i+1}, R_{i+1})$
S2	$l(C_{i+1}, R_{C'_i})$
S3	$l(C_{i+1}, R_{C'_i} \cup R_{C''_i})$ and $!l(C'_i, R_{C'_i})$ and $!l(C''_i, R_{C''_i})$
S4	$l(C_{i+1}, R_{m_i})$ and $!l(C'_{i+1}, R_{m_i})$
S5	$l(C_{i+1}, R_{C'_i})$
S6	$l(C_{i+1}, R_{C'_i})$
S7	$!l(C_i, R_{C_i})$
S8	$l(C'_{i+1}, R_{C_i})$ and $l(C''_{i+1}, R_{C_i})$ and $!l(C_i, R_{C_i})$
S9	$l(C''_{i+1}, R_{C_i})$ and $l(C''_{i+1}, R_{C'_i})$ and $!l(C_i, R_{C_i})$ and $!l(C'_i, R_{C'_i})$
S10	$l(m_{i+1}, R_{i+1}) \mid Sim(m_{i+1}, R_{i+1})$
S11	$l(m_{i+1}, R_{m'_i})$
S12	$l(m_{i+1}, R_{m'_i} \cup R_{m''_i})$ and $!l(m'_{i+1}, R_{m'_i})$ and $!l(m''_{i+1}, R_{m''_i})$
S13	$l(m_{i+1}, R_{m'_i})$ and $l(m''_{i+1}, R_{m'_i})$ and $!l(m'_i, R_{m'_i})$
S14	$!l(m_i, R_{m_i})$
S15	$l(m'_{i+1}, R_{m_i})$ and $l(m''_{i+1}, R_{m_i})$ and $!l(m_i, R_{m_i})$
S16	$l(m''_{i+1}, R_{m_i})$ and $l(m''_{i+1}, R_{m'_i})$ and $!l(m_i, R_{m_i})$ and $!l(m'_i, R_{m'_i})$
S17	$l(m_{i+1}, R_{m_i}) \mid Sim(m_{i+1}, R_{m_i})$
S18	$!l(m_i, R_{m_i}) \mid \nexists Sim(m_{i+1}, R_{m_i})$
S19	$l(C'_{i+1}, R_{C_i})$ and $!l(C_{i+1}, R_{C_i})$
S20	*null*
S21	$l(C'.m_i, R_{m_i})$ and $!l(C.m_i, R_{m_i})$
S22	$l(C'.m_i, R_{m_i})$ and $!l(C.m_i, R_{m_i})$
S23	$l(C'.m_i, R_{m_i})$ and $!l(C.m_i, R_{m_i})$
S24	$l(m'_{i+1}, R_{m_i})$ and $!l(m_{i+1}, R_{m_i})$

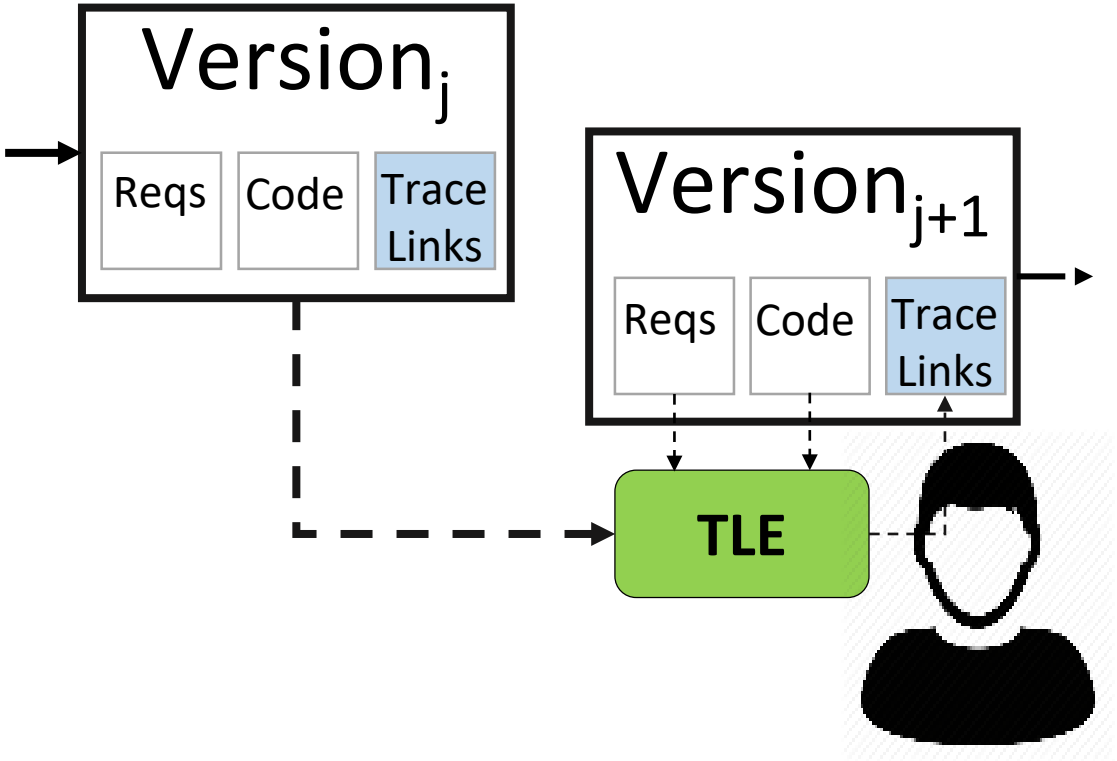
② Infer which change scenario was enacted.

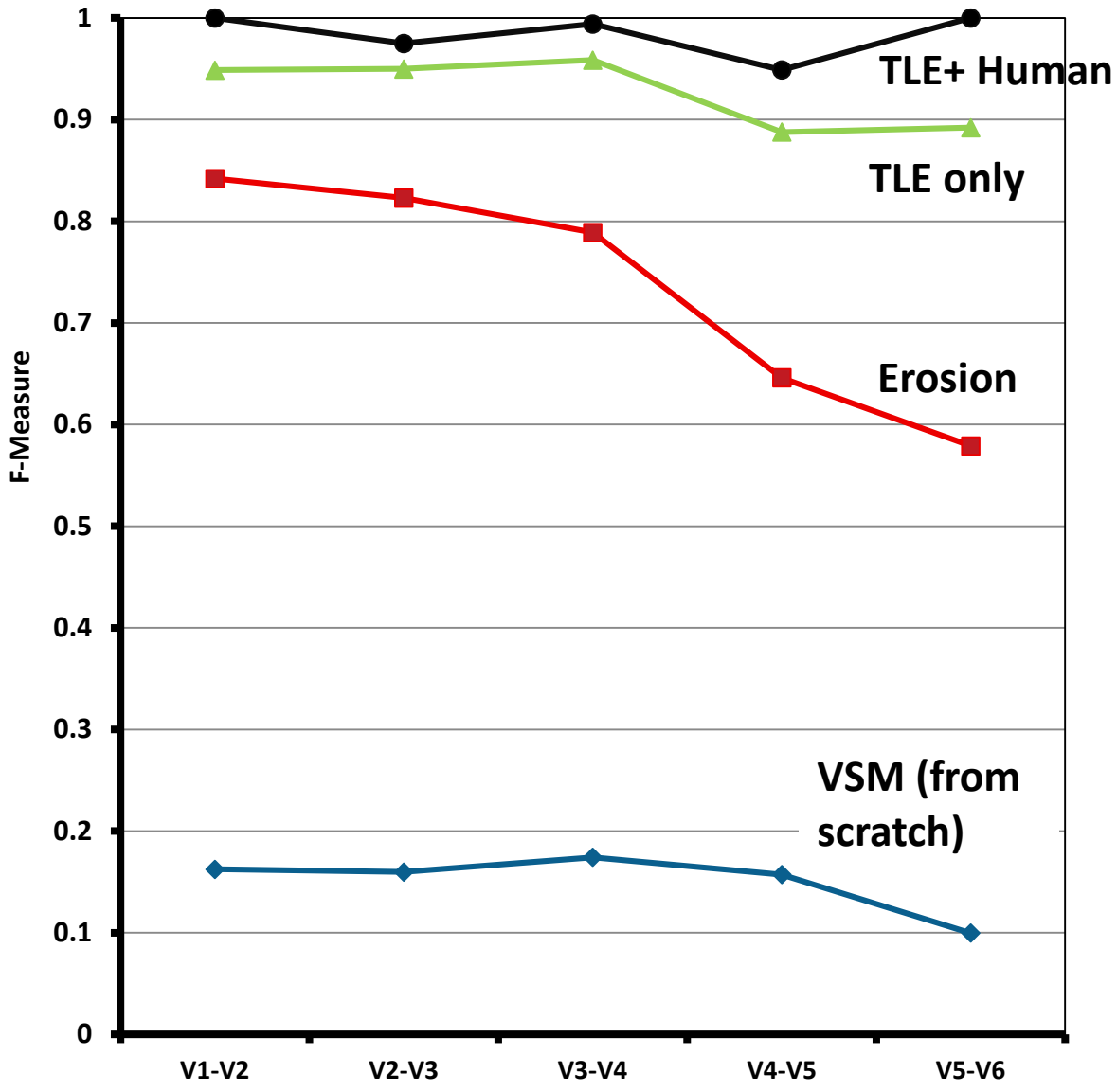


S_3 The new class represents a merging of existing ones. Reassign any trace links that associated with the original classes to the merged class.

③ Update links

Human in the loop



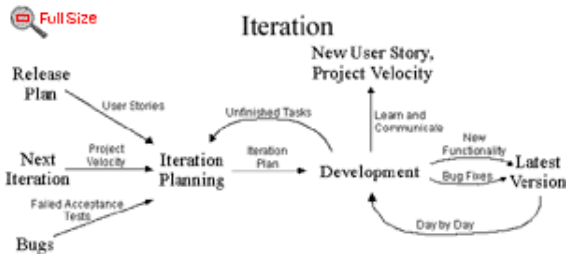


In our experiments the effort required by humans to confirm or deny TLE links was minimal – with few decision points per day.

A Retrospection on factors that impacted our decisions

Made an explicit decision to favor agility.

Product size



Considered uncertainty, volatility, and velocity.

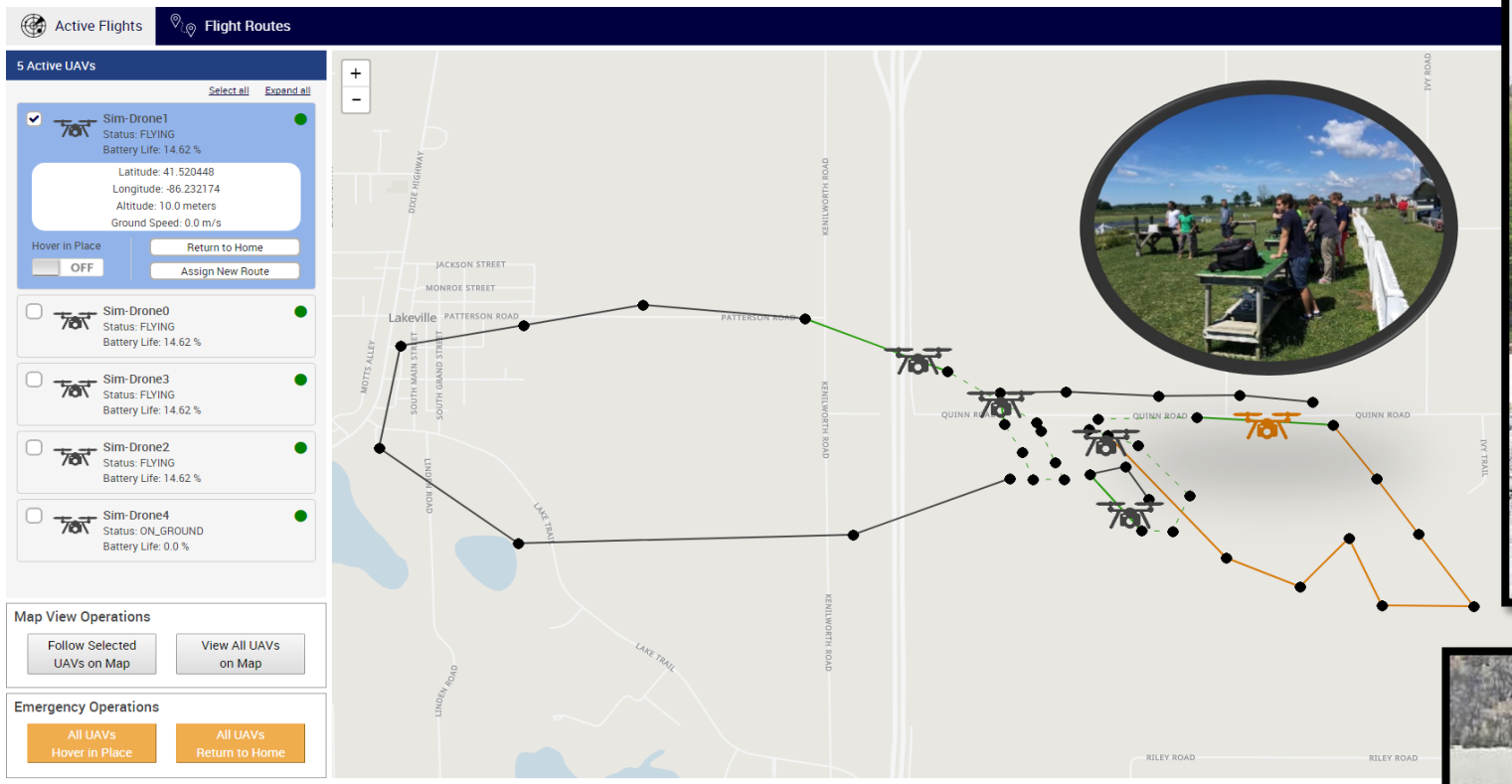
Made timeline decisions based on multiple factors:
-- time to “market”
-- difficulty
-- current workload

Some open research challenges

- Merging mining software repositories (jira and github etc) with static analysis, runtime traces, and topic modeling to identify potential features.
- Providing automated help for transforming tangled code into a variability point (either static or runtime).
- Human Computer Interaction: Creating a dashboard to serve as a burndown chart to show current progress (technical debt) towards achieving variability points.
- “Real” impact analysis where I can create a ‘user’ story defining a new variability point – and the analysis identifies point of impact and difficulty of transforming the code and supporting artifacts.

Dronology Project @ Notre Dame

http://Dronology.info



A platform for coordinating the flight of UAVs. Supports research in safety assurance, runtime monitoring, & adaptation.

Developed by the Notre Dame Team: Michael Vierhauser, Jane Wyngaard, Jinghui Cheng, Sean Bayley, Greg Madey, Joshua Huseman, Jane Cleland-Huang, & more…

Any questions?

Reverse Engineering Product Lines in Agile Environments

Jane Cleland-Huang, PhD

Software Product Line Conference, Seville, Spain 2017

JaneClelandHuang@nd.edu

Department of Computer Science and Engineering

University of Notre Dame

Some of the work described in this talk was funded by the US National Science Foundation under Grants CCF-0959924 and CCF-1265178.